GigaDORAM: Breaking the Billion Address Barrier

Abstract

We design and implement GigaDORAM, a novel 3-server Distributed Oblivious RAM (DORAM) protocol. Oblivious RAM allows a client to read and write to memory on an untrusted server, while ensuring the server itself learns nothing about the client's access pattern. Distributed Oblivious RAM (DORAM) distributes the role of the ORAM server. Specifically, DORAM allows a group of servers to efficiently access a secret-shared array at a secret-shared index.

DORAM has two main advantages over traditional ORAM protocols: (1) it effectively outsources all the communication / computation costs to the DORAM servers, minimizing the client complexity, and (2) it allows secure multiparty computation (MPC) in the RAM model, opening a new range of potential applications.

A recent generation of DORAM implementations (e.g. FLORAM (CCS '17), DuORAM (ePrint 2022)) have focused on building DORAM protocols based on Function Secret-Sharing (FSS). These protocols have low communication complexity and low round complexity but high linear computational complexity. Thus, they work for moderate size databases, but at a certain size these FSS-based protocols become computation-bound and performance degrades dramatically.

In this work, we introduce GigaDORAM, a hierarchical-solution-based DORAM which leverages several novel round-reducing tricks. GigaDORAM features poly-logarithmic computation and communication which is comparable to that of other hierarchical based (D)ORAMs, but with an over $45\times$ reduction in rounds per query. In our implementation, we show that for small databases GigaDORAM is slightly faster than FSS-based schemes, but for moderate to large databases where FSS-based solutions become computation bound, our protocol is orders of magnitude more efficient than the best existing DORAM protocols. At $N = 2^{19}, 2^{25}, 2^{31}$ we achieve over $10\times, 160\times, 2500\times$ improvement to queries/sec over state-of-the-art, respectively. When $N = 2^{31}$, our DORAM is able to perform over 700 queries per second while previous constructions could not handle a single query per second.

1 Introduction

To an outside observer, traditional encryption schemes can effectively hide the *contents* of a memory, yet encryption alone does not hide the memory locations being accessed. In many cases, the *access pattern* of a file system can leak sensitive information, even when the contents are encrypted.

Oblivious Random Access Memory (ORAM) [22,37,38] is a cryptographic protocol that allows a client to read and write[1] from memory while ensuring the *physical access pattern* (which is potentially observable to someone with sufficient access to the machine) is independent of the *virtual access pattern* (the underlying data retrieved by the client). Thus, when memory is accessed using an ORAM protocol, it is *mathematically provable* that an observer learns nothing about the client's query pattern (beyond the number of queries).

Oblivious RAM was developed in a model where a single client wishes to store and retrieve sensitive data from an untrusted data store. Originally, the untrusted data store was conceptualized as untrusted RAM on the same machine as the client, but today we usually imagine a client storing and retrieving data from an untrusted cloud provider (See Figure 1). In this setting, encryption can hide the *data* from the cloud provider, but ORAM is necessary to hide the *access pattern*[2]. Thus, ORAM provides the strongest possible guarantee – hiding *both* the data and the access pattern.

Although ORAM was designed in the client-server setting, ORAM is also useful in the context of secure multiparty computation, where a group of servers need to access a secret-shared array at a secret-shared location. In this setting, the

[1] This is in contrast to Private Information Retrieval (PIR) protocols, that only allow reads.

[2] Note that most Searchable Symmetric Encryption (SSE) schemes allow a client to efficiently query encrypted data stored in an untrusted cloud, but they typically do *not* hide the access pattern from the cloud provider. SSE schemes also target a different model, where data payloads may be of drastically different size (in ORAM all entries are of size D) and queries may return different number of "matches" [34]

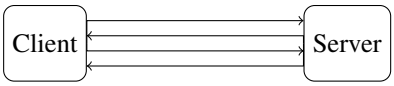

Figure 1: Client-Server ORAM. In this model we minmize client-server communication.

secret sharing hides the *data*, but every participating server observes the physical access pattern. *Distributed Oblivious RAM (DORAM)* provides a method for efficiently accessing a secret-shared array at a secret-shared index, which in turn makes it possible to do secure multiparty computation (MPC) in the RAM model (RAM-MPC). Almost all existing MPC protocols work in the *circuit-model* where the desired computation is first converted to an arithmetic/boolean circuit before being executed securely. This conversion can be extremely costly (e.g. in the case of private database) because every "random-access" is replaced with an $O(\text{memorySize})$ "MUX operation." RAM-MPC allows random-access programs to be executed securely without this costly conversion, which in turn enables much more flexible and efficient secure multiparty computation protocols.

In this work, we focus on Distributed Oblivious RAM (DORAM). Broadly speaking, we focus on DORAM because in the client-server setting, DORAM allows (*many*) clients to access memory in a single round of client-server interaction. Moreover, in the RAM-MPC setting, DORAM allows for *significantly* more efficient secret-shared memory access than traditional ORAM. Below, we compare DORAM and ORAM in both settings, explore the DORAM state-of-the-art, and outline our contributions.

1.1 DORAM vs. ORAM in the client-server setting

Although DORAM protocols are usually described as having no client, DORAM can be turned into a client-server ORAM by having a separate client secret-share their query to the servers. That is, the DORAM servers hold secret shares of an array $[\![x_1]\!], \ldots, [\![x_N]\!]$, the client sends secret shares of an index i, $[\![i]\!]$, to the servers, and the servers respond with shares $[\![x_i]\!]$ which the client can combine to get x_i. With no comparative efficiency loss, the client can write to the secret-shared array such that the servers do not learn what was written and to where. (See Figure 2).

Thus, leveraging the generic transformation above, a DORAM can substitute for an ORAM in the client-server setting. Moreover, DORAM can make for *an improvement* over ORAM in the client-server setting because it supports many extremely lightweight, not necessarily co-located clients while ORAM can only reasonably support a single LAN-connected client. Specifically, DORAM has the following benefits in the client-server setting:

Client rounds-per-query: In hierarchical ORAMs, to query

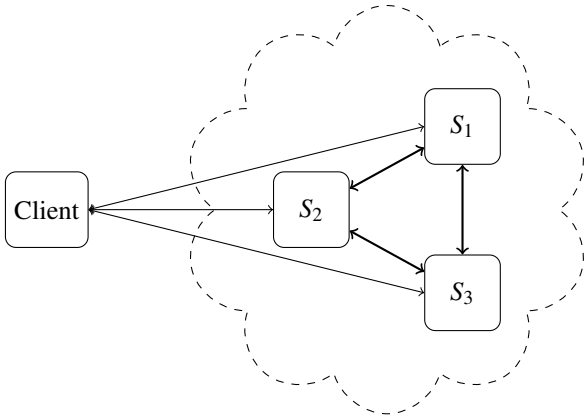

Figure 2: Using 3-party DORAM to simulate a client-server ORAM. Note that DORAM measures the cost of communication between the 3 DORAM servers. The client secret shares an index ($3\log(N)$ bits of communication), and receives a data payload ($3D$ bits of communication). Thus the communication between the client and the DORAM servers is always efficient. Thus, we focus on minimizing the *server-to-server* communication.

an ORAM server a client must participate in $O(\log N)$ sequential rounds of communication *per query* and occasionally must participate in much more expensive "rebuilds." On the other hand, in a DORAM the client need only communicate $[\![i]\!]$ to the servers. In practice, when the client and server may not be co-located (as is often the case in client-server settings), this makes a *tremendous* difference. To illustrate, assuming that the constant on $O(\log N)$ is 5 (often much higher) and there is 70ms[3] latency to the ORAM server, at $N = 2^{25}$ we get that ORAM takes at least $8.75s$ to perform a single query. Meanwhile, in this setting, GigaDORAM requires approximately 72ms to complete a query (including latency to receive query and send response) – an improvement of over $100\times$.

Client-server communication: The communication between the client and the DORAM servers is independent of the servers' work and scales only with n, the number of servers. When there are three DORAM servers, this means that the client's query size is only $3\times$ the cost of a query in the insecure setting[4]. In fact, this gives *lower* client-server communication than the $(O(\log N(\log N + D))$ bits-per-access) lower bound in the traditional client-server ORAM model [22, 29]. Each client query results in significant *server-to-server* communication costs, but this may be reasonable when the servers have much higher bandwidth and lower latency connections to each other than to the client. Although the client-to-servers communication is always low in this model, in this work, we

[3] The lowest ping we measured to the nearest AWS region was 70ms.

[4] In a 3-party DORAM, a client's query can be a secret-shared index ($3\log N$ bits), a data element for write queries ($3D$ bits), and a boolean flag determining whether the operation is a read or write, so the total query cost is $3(\log N + D + 1)$ bits.

show that the server-to-server communication can be made quite low as well.

Multiple clients: A client-server ORAM protocol can only serve a single client. On the other hand, when using DORAM to simulate an ORAM server (c.f. Figure 2), the system is immediately multi-client, because the client is completely stateless. Moreover, using lightweight MPC techniques, the servers could enforce private access control policies on *private* credentials (as in [17]).

Cost of rebuilds: Many client-server ORAM protocols only achieve *amortized* communication complexity, and after a certain number of queries, an extremely expensive "rebuild" step is required to preserve privacy. For a real world client, this means that after a certain number of queries the client might need to wait minutes before she can execute her next query, which is often unacceptable. Although there are ORAM protocols with asymptotically low worst-case communication complexity (e.g. [4]), they are not efficient in practice. In DORAM protocols, rebuild costs are incurred only by the servers with no client involvement. DORAM rebuilds tend to execute faster in practice due to the servers strong compute and good server-to-server connections.

Client code-complexity. In traditional client-server ORAM, the client code is fairly complex. for example in [44], a work aimed at creating a low-complexity client, their client software requires over 4000 lines of C++. By contrast, our DORAM client is sufficiently simple to be written in less than 20 lines of in-browser plain Javascript (which we provide).

Trust Assumptions: Of course, the downside of using DORAM to simulate a single ORAM server is that it slightly strengthens the trust assumptions. A (3,1)-DORAM protocol will only be secure if 2 out of the 3 of the DORAM servers are honest, whereas a traditional client-server ORAM protocol (with only a single server) remains secure even if the sole server is dishonest.

1.2 ORAM vs. DORAM in the RAM-MPC setting

Although ORAMs were traditionally designed for the client-server setting, they can be also used to enable RAM-MPC. In particular, given an ORAM scheme and an MPC framework, a set of servers can simulate an ORAM client under generic MPC, effectively compiling the ORAM to a DORAM protocol that can be used for RAM-MPC. Although this generic transformation yields a provably secure DORAM protocol, the resulting protocol may not be *efficient*. The bottleneck arises because the ORAM client code may not be "MPC-friendly," i.e., simulating the ORAM client under MPC may incur large losses in efficiency. To address this problem, ORAM protocols like Circuit ORAM [46] were designed with the goal of minimizing the *circuit complexity* of the ORAM client, thus making it easier to simulate the ORAM client under MPC.

By contrast, DORAM is a natural choice for implementing RAM-MPC because the MPC clients are readily available to play the role of DORAM servers, and no additional participants or trust assumptions are necessary. As we show in Section 10, our DORAM protocol, which is custom-built for the multiparty setting, can be made *much* more efficient than protocols which were designed in the client-server setting, and generically "compiled" into a DORAM protocol.

1.3 Previous DORAMs

Although there are many different ways to measure the efficiency of a (D)ORAM protocol, since the formalization of (D)ORAM 30 years ago [37–39] the theory community has emphasized the (amortized) communication complexity per query. After years of research, several DORAM protocols that have small ($O(\log(N))$) communication overhead have been presented (e.g. [19, 32]). Unfortunately, these protocols have high round complexity, requiring $O(\log(N))$ rounds of communication (with constants usually ranging $80-150$) for every query. In practice, network latency is a big bottleneck for performance, and consequently low-communication high-round (D)ORAM protocols have seldom been implemented. These communication-optimal protocols are built using the *"hierarchical solution"* [37, 38], which we describe in Section 3.

It has been noted that the high round complexity of the hierarchical solution makes it unsuitable for practical applications [48], so most DORAM *implementations* (e.g. [8, 16, 24, 26, 43, 45, 46, 48, 53]) take a completely different approach. These constructions focus on minimizing rounds while compromising on either asymptotic computation or asymptotic communication costs. One common technique (most recently applied by DuORAM [45]) for creating DORAM protocols with low round complexity *and* low communication complexity is Function Secret Sharing (FSS) [7, 20]. Unfortunately, while FSS results in low *communication* complexity, FSS-based protocols generally require $O(N)$ *computation* per query. Thus, protocols like DuORAM shine in high-latency, low-bandwidth networks. Yet, as we show in Section 10, in less constrained networks these protocols quickly become *computation* constrained, and cannot scale to large values of N. Additionally, as we argue in Section 10, while it is impressive that a DORAM protocol could be built to operate in such constrained network settings, these settings are too slow for MPC and hence not the networks where one would deploy RAM-MPC. Additionally, we argue that these networks are worse than what is available for non-collated servers.

1.4 Our Contributions

In this work, we design and implement a high-performance DORAM protocol in the (3,1)-security model, i.e., where

there are three semi-honest servers, and no two of them collude. With our construction, we bridge the gap between asymptotically-low overheads and few rounds per query. This allows us to scale well beyond the limits of where FSS-based (D)ORAM protocols like FLORAM and DuORAM become computationally constrained.

To achieve this, we base our novel DORAM protocol on the hierarchical solution, which immediately gives us low communication complexity, low computational complexity, but high round complexity between the servers. Thus, in this paper, we focus on decreasing the number of rounds per query. To do so we present several novel optimizations that dramatically cut rounds for (D)ORAMs based on the hierarchical solution.

Asymptotically we present a DORAM which matches the best-known [3, 4, 19, 32] $O((\kappa+D)\log N)$ communication / computation per query and another which has slightly worse asymptotics ($O(\kappa^2+D\log N)$ communication per query) but saves over 35 rounds per query. Roundwise, our DORAM requires only 35 rounds per query, a 45-fold improvement from the previous most round-efficient hierarchical DORAM [19].

The implementation of our DORAM protocol required over 8,000 lines of custom C++ code. Once the paper is deanonymized, we will contribute our implementation to EMP-Toolkit [47], a popular, actively maintained, high-performance, *open-source* MPC implementation library. Our contribution will wrap DORAM in a simple 2-function (Initialize and Query) API which allows users to enable RAM-MPC without a deep understanding of the code. Additionally, we provide lightweight client implementation in Python and JavaScript, enabling users to to utilize DORAM in the client-server (Figure 2).

In addition to the implementation of our DORAM protocol, we contribute (1) a from-scratch competitive implementation of the [2] general MPC framework which, in many settings, is the fastest known 3-party MPC protocol.[5] (2) A custom (3,1)-garbled circuits protocol built using EMP-toolkit's 2-party garbled circuits, which can be imported separately from ABY3's [42] giant framework, and (3) The first tested circuit files of the LowMC block cipher [1], featuring a novel optimization which reduces cache misses.

2 Real-world motivations

Private Contact Discovery: The popular secure-messaging app Signal uses Path ORAM [44] for contact discovery [12]. In Signal, each user account is linked to a 10-digit phone number, meaning the Signal user database is indexed by (at most) $N = 10^{10} \approx 2^{32}$ keys. The first time a Signal user wishes to send a message to a contact, the user must search this database using the contact's phone number.

Signal uses Path ORAM to hide the user's query from Signal's own servers. The problem with this approach is that Path ORAM works in the client-server model, so it only supports a single client.

To get around this problem, Signal implements a single ORAM client inside an Intel SGX enclave. When a user makes a query, the user passes the query term (i.e., the phone number) to the SGX enclave, and the SGX enclave makes the ORAM query. Since the SGX enclave is acting as the trusted ORAM client, and the machine's main memory is acting as the ORAM server, the client and server are on the same physical machine, thus the bandwidth is extremely high, and the latency is extremely low, allowing them to scale up to an ORAM with $N > 2^{30}$. While impressive, by using an Intel SGX to drive their ORAM, Signal reduces the cryptographic security of ORAM to the unproven security the enclave.

Alternatively, one could use DORAM to achieve the same multi-client goals *without* requiring trusted hardware to act as the single ORAM client. As we will see in Section 10, our cryptographically-secure DORAM can scale to $N > 2^{30}$ even when the DORAM servers are not on the same physical machine.

Private Bulletin Boards: Riposte [13] is an anonymous messaging system, where each user has a "mailbox" and users can deposit and retrieve messages in specific mailboxes without revealing whose mailbox was accessed. The system uses a two-server PIR protocol (with writing) based on Function Secret Sharing (FSS) [7, 20]. The drawback of Riposte is that although the communication between clients and the servers is low, the server-side computation is *linear* in the number of users, N. A similar system could be built using DORAM, decreasing the server-side computation from $O(N)$ per query, to $O(\log N)$ per query (amortized).

Private, Account-Based Ledgers: Another application of DORAM is to improve privacy in the blockchain space. On most blockchains (e.g. Ethereum, Algorand, Avalanche, Solana) user balances are stored in cleartext. One natural way to improve privacy is to store user balances in an ORAM. Thus, a transfer would be two ORAM queries, one to decrement the sender's balance and one to increment the receiver's balance. Simple Zero-Knowledge Proofs (ZKPs) could be used to ensure that these updates follow the stated rules of the blockchain (e.g. the amount decremented from the sender is equal to the amount incremented for the receiver). Solidus [9] provided a proof-of-concept for a privacy-preserving blockchain like this. Inherently distributed, DORAM is perfectly suited to this setting. The blockchain validators would play the role of the DORAM servers, and since a trusted threshold of validators is already needed for consensus, there would be no additional trust assumptions in having them play the role of DORAM servers[6]. This setting is also inherently

[5] The implementation used to benchmark the results in the [2] is proprietary, and not publicly available.

[6] Although Solidus is only a proof of concept, several real-world blockchains (e.g. Axelar, Thorchain) use their validator sets to run MPC

multi-client, so traditional client-server ORAM is inapplicable.

3 Preliminaries

Notation. We let N be the number of elements in the DORAM database, D the size in bits of each element, and κ be the computational security parameter (in practice $\kappa = 128$, in theory $\kappa = \omega(\log N)$), σ the statistical security parameter (in practice 2^{-40} or 2^{-80}).

Secret sharing. Our DORAM protocol makes heavy use of cryptographic secret-sharing. Throughout this work, we use $[\![\cdot]\!]$ to denote a "replicated" (or CNF [14]) secret sharing. In a 3-party replicated sharing, a secret, x is split into three shares $x = x_1 \oplus x_2 \oplus x_3$, and participant x gets *two* of the shares – every share except x_i. We use $[\cdot]^{(i,j)}$ to denote a simple XOR-2-sharing between participants P_i, P_j

Obliviousness: A computation is data-*oblivious* if its control-flow is independent of the input data. Many multiparty computation models work in the circuit model of computation, because the control flow of a circuit-based computation is determined before data is input to the computation and hence, computations in the circuit model are inherently oblivious. An Oblivious RAM protocol is a protocol for accessing an array (indexed by $1,\ldots,N$), where the algorithm's control flow, and in particular, the physical memory accessed, is independent of the index being queried. ORAM protocols are designed to allow a client to make a *any sequence* of queries and are often composed of simpler data structures which are only oblivious on distinct queries. A data structure (e.g. a hash table) is called *distinct-query oblivious* if the control-flow between any two sequences of *distinct* queries is indistinguishable, but a sequence with repeated queries might result in a control flow that is distinguishable from a sequence of distinct queries.

OHTable, OSet. We call an efficient, oblivious to build, distinct-query oblivious, hash table an *OHTable*. Similarly, we call an efficient, oblivious to build, distinct-query oblivious, set data structure an *OSet*. When working in the multi-server setting (as we do), OHTables/OSets often have interactive, distributed, and secret-shared components. Oblivious Hash Tables have three key functionalities: Build, Query and Extract. $\mathcal{F}_{\text{OHTable}}.\text{Build}(X)$ creates an oblivious hash table storing the elements X. Once the hash table has been built, $\mathcal{F}_{\text{OHTable}}.\text{Query}$ queries the table, and $\mathcal{F}_{\text{OHTable}}.\text{Extract}$ extracts all elements currently stored in the table which have *not* been queried.

The hierarchical solution. The hierarchical solution [38] is a compiler which converts a distinct-query OHTable into a full-fledged ORAM protocol by creating a hierarchy of $O(\log N)$ OHTables of geometrically increasing size. The hierarchical solution is a powerful tool, used to build many ORAM protocols, e.g. [3, 19, 22, 23, 28, 32, 37, 38, 40].

A hierarchical ORAM is made up of $L_0,\ldots,L_{\mathsf{numLevels}}$ where L_0 is a special "cache", usually implemented as an append-to-write, read-all-to-read table . This makes reading from and writing to the cache oblivious, but inefficient. Because reading from the cache is inefficient, the cache is usually set to be constant size (i.e., $|L_0| = O(1)$). For each $i \in \{1,\ldots,\mathsf{numLevels}\}$, the level L_i is usually instantiated as a distinct-query OHTable of size $O(2^i)$.

The major innovation of the hierarchical solution is its query pattern, which periodically rebuilds the levels of the hierarchy to guarantee that no element is ever queried at the same level twice between rebuilds. The rebuild functionality uses $\mathcal{F}_{\text{OHTable}}.\text{Extract}$ to extract all elements in a given level, and rebuild them into a new table at a lower level in the hierarchy. Since no element will be queried twice in any given distinct-query OHTable, the entire construction is oblivious, even if queries are repeated. The basic idea of the rebuild schedule is that level L_i is rebuilt every $O(|L_i|)$ queries, and every rebuild costs roughly $O(|L_i|)$, $O(\mathsf{numLevels})$, where commonly $\mathsf{numLevels} = O(\log N)$ and the largest level, $L_{\mathsf{numLevels}}$,has size $O(N)$.

DORAM. A distributed Oblivious RAM protocol is a multiparty protocol that allows a group of participants holding a secret-shared array $[\![x_1]\!],\ldots,[\![x_N]\!]$ to access the array at a secret-shared index, $[\![i]\!]$, and obtain the sharing $[\![x_i]\!]$, without revealing any information about the query, i, or the database $x_1,\ldots,x_n$. The theoretical efficiency of DORAM protocol is often measured by the amortized communication complexity the servers spend to respond to a single query. In practice, DORAM protocols may be bottlenecked the amortized communication per query (e.g. [46]), the amortized computation per query (e.g. [16, 45]), or the amortized number of communication rounds per query (e.g. [19, 32]). To compare between these constructions targeting different points in the solution space, *we measure the practical efficiency of a DORAM protocol by the number of queries per second that it can process.*

SISO-PRFs: A core building block of most DORAM protocols is a Shared-Input, Shared-Output PRF (SISO-PRF). A SISO-PRF allows the participants to compute the secret-sharing of a PRF output on a shared input, under a shared key. Any regular PRF can be converted into a SISO-PRF by implementing the PRF under a generic MPC protocol. Several PRF protocols have been designed to be "MPC friendly" (e.g. LowMC [1]). The basic idea which makes SISO-PRFs useful for DORAM is that servers can generate a random, shared key, $[\![k]\!]$, and build a hash table where the cleartext tags (of secret-shared payloads) are SISO-PRF evaluations of the elements.

4 Secure Multiparty Computation

Secure multiparty computation (MPC) is a protocol that allows a group of participants to securely compute a joint func-

protocols (specifically Threshold Secret Sharing), so it is not a big stretch to think a blockchain validator set could run a DORAM protocol as well.

tion on their private inputs *without* revealing any information beyond the output of the function. An MPC protocol is called (n,t)-secure if the protocol involves n participants, and remains secure if at most t participants collude (i.e., share private state). MPC has been widely studied [11, 21, 50, 51], and many reasonably efficient MPC protocols are known. Our DORAM protocol works in the (3,1) model which assumes there is no collusion between the servers. We use the (3,1)-"replicated" MPC protocol of [2] as a building block. In one crucial place, we also use the (3,1) garbled circuit MPC protocol of ABY3 [33] to reduce round complexity. Our DORAM protocol is also an important tool in *building* efficient RAM-MPC protocols, because it allows for MPC computation in the *RAM model* of computation, whereas most current MPC protocols work in the *circuit model*.

4.1 The Arithmetic Black-Box Model

Our DORAM protocol makes use of several "basic" operations on secret shared values, e.g. addition, comparisons, and equality tests. In our protocol descriptions, we use the Arithmetic Black Box (ABB) model to abstract away the underlying implementations of these operations. In practice, we use our own implementation of the extremely efficient 3-party MPC protocol of [2]. A complete, formal description of the ABB model can be found in [19, 25].

In protocols, we use our ABB by invoking $\mathcal{F}_{\text{ABB}}$.FunctionalityName, where FunctionalityName makes it obvious what the functionality achieves. For instance, we invoke $[\![z]\!] = \mathcal{F}_{\text{ABB}}.\text{Mult}([\![x]\!],[\![y]\!])$ to multiply secret shared values x, y and obtain secret shared value z s.t $z = x \cdot y$. Although the names are usually self-explanatory, we provide a complete list of our ABB functionalities in Appendix A.

5 Construction Overview

The starting point for our construction is the (3,1) semi-honest DORAM of [19], which uses the hierarchical solution in a distributed setting to achieve amortized communication and computation per query of $O((\kappa + D)\log N)$[7]. Although [19] has great *communication* overhead, the *round-complexity* of that protocol makes it inefficient in practice. In this work, our goal is to draw inspiration from [19] to maintain low communication and computation overhead while *significantly* reducing the round complexity.

One problem with the hierarchical solution is that every query to the DORAM forces a query into *every* level of the hierarchy, $L_0, \ldots, L_{\mathsf{numLevels}}$. Critically, these queries must be *sequential*, because if the item is found at L_i, the protocol must query dummy elements at $L_{i+1}, \ldots, L_{\mathsf{numDummies}}$. Thus the *round complexity* of a DORAM query[8] , numRoundsDORAM, can be written as

$$\mathsf{numRoundsOHTable} \cdot \mathsf{numLevels} + \mathsf{numRoundsCache} \quad (1)$$

where numRoundsOHTable is the number of rounds to query L_i for $1 \leq i \leq$ numLevels (which is fixed and independent of i) and numRoundsCache is the number rounds it takes to query the cache, L_0.

Outline of objectives: In light of Equation 1, we can divide our efforts to reduce round complexity into four parts: (1) *reducing* numRoundsOHTable *via our novel OHTable, ShufTable, and SISO-PRF parallelization*, (2) *generalizing the hierarchical-solution with a tunable parameter,* baseAmpFactor*, to reduce* numLevels, (3) *designing a new L_0-cache data-structure, SpeedCache, to reduce* numRoundsCache, and (4) *applying additional engineering-level optimizations.*

With these design improvements, we are able to significantly reduce the round complexity of the hierarchical solution, while maintaining similar asymptotic overheads to the theoretic state-of-the-art, [19]. In addition to its low asymptotic complexity and low rounds-per-query, we show that our DORAM design is actually quite fast in practice (Section 10).

We briefly explain each of our optimizations below and expend on them in Section 6, 7 and 8 respectively. In addition to these structural changes to the protocol, we employ several crucial implementation-level optimizations which we describe in Section 9.

(1) Reducing numRoundsOHTable**: ShufTable & SISO-PRF parallelization.** To reduce numRoundsOHTable we present a novel, standalone OHTable, called *ShufTable*, with dramatically reduced round complexity over the OHTables used in previous hierarchical ORAM constructions. To further round complexity, we devise a method to parallelize the sequentially-dependent SISO-PRF evaluations needed to query each level of the hierarchy.

The key ideas in these optimizations revolve around how to handle queries for elements that *are not in the table*. This type of query happens frequently in a hierarchical ORAM, because although each element is only stored at one level of the hierarchy, each ORAM query results in a query to the OHTable at every level of the hierarchy.

To address this, [19] developed a novel OSet construction, to efficiently determine whether the desired index was stored in the table. When the index was *not* stored in the OHTable, the protocol would search the table for a precomputed "dummy" index. Although this approach can be made very communication efficient, it requires many *rounds* of communication, so we take a new approach.

[7] [19] was not the first to achieve these asymptotics; [32] achieved the same asymptotics but the hidden constants were much higher.

[8] DORAM also incurs round costs when building a level L_i and extracting from a level L_i. Since the protocols are only invoked every $|L_i|$ queries and have small, constant, round costs (where our final protocol has $|L_0| = O(\kappa)$), their round cost has negligible impact on the performance of DORAM.

Our novel OHTable, ShufTable still inserts dummy elements $(d_1, \perp) \ldots, (d_l, \perp)$ along with the real elements, and retrieves a dummy when the queried element is not stored in the table. We differ from [19], however, in that we eliminate the OSet, and instead use a "just-in-time" mechanism to detect if an element is stored in the table, and replace the retrieved element with a dummy if necessary. Assuming the SISO-PRF has already been evaluated, ShufTable is queriable in just 5 rounds of server-to-server communication. The main ingredient that enables this round-savings is a novel "persisted shuffling" trick (Section 6.1) which allows to efficiently evaluate a random permutation under MPC. Our OHTable, like those in prior DORAM constructions, requires an equality-check on secret-shared values. We implement this equality check using a custom implementation of a 3-party garbled circuit [33]. This *increases* the asymptotic communication of each level's query from $O(\kappa)$ to $O(\kappa^2)$, but *decreases* the round complexity at each level by $O(\log \kappa)$. In practice, when $\kappa \in \{128, 256\}$, this dramatically improves real-world performance. If one were focused on optimizing asymptotic communication complexity, this step could be replaced by an MPC-based equality check (e.g. using [2]), which would make our asymptotic communication overhead the same as [19]. More details can be found in section 6.2 and the protocol is given in Figure 6.

Second, leveraging a feature of ShufTable, we reduce $\mathsf{numRoundsOHTable}$ by observing that we can the parallelize all the SISO-PRF evaluations we need for each level of the hierarchy. In particular, like in [19], we use SISO-PRF evaluations as clear-text tags for secret-shared elements while building and retrieving from our Cuckoo hash tables. In the hierarchical construction, the query into table L_{i+1} depends on the result of the query into L_i. Thus, hierarchical based (D)ORAMs wait until after L_i.Query is performed to evaluate the SISO-PRF for L_{i+1}. In Section 6.3 we show how to circumvent this limitation and evaluate the PRFs for all levels at once, saving $(\mathsf{numLevels} - 1) \cdot \mathsf{numRoundsPRFEval}$ rounds per query. In practice this is an enormous savings, saving ~ 100 rounds of interaction per query. To further reduce rounds, in practice, we parallelize the query of the cache, L_0 with the evaluation of the PRFs for $L_1, \ldots, L_{\mathsf{numLevels}}$ (Section 9). Note that if we knew multiple queries in advance, we would be able to further batch evaluations, but we do not assume that in this work.

Overall, these optimizations have a significant impact of the efficiency of our protocol. We decrease $\mathsf{numRoundsOHTable}$ from the 45 of [19] to amortized $5 + \mathsf{numRoundsPRFEval}/\mathsf{numLevels} \approx 7$. Additionally, we reduce the number of expensive SISO-PRF evaluations necessary by a factor of four when compared to [19]'s OSet-driven approach.

(2) Generalizing the hierarchical solution to reduce $\mathsf{numLevels}$. Above, we described our techniques for reducing the round complexity of queries to individual OHTables. Yet in the hierarchical solution, every ORAM query requires querying each level of the hierarchy *sequentially*. Thus a hierarchy of depth $\mathsf{numLevels}$ immediately adds a multiplicative factor of $\mathsf{numLevels}$ to the round complexity of the protocol. Since round complexity is one of the main performance bottlenecks in ORAM protocols, there is a strong motivation to reduce $\mathsf{numLevels}$.

In the original hierarchical construction, and all subsequent constructions of which we are aware, level i in the ORAM hierarchy had size $2^i \cdot |L_0|$, resulting in $\mathsf{numLevels} = O(\log_2 N)$. In Section 7 we show that by introducing a new parameter, $\mathsf{baseAmpFactor} > 2$, and setting $|L_{i+1}| = \mathsf{baseAmpFactor} \cdot |L_i|$, we can dramatically reduce the *round complexity* of the protocol with minimal impact on the *communication* complexity. This simple change immediately reduces $\mathsf{numLevels}$ from $\log_2(N)$ to $\log_2(N)/\log_2(\mathsf{baseAmpFactor})$. While this modification is conceptually simple, it requires a more nuanced rebuild schedule.

Somewhat surprisingly, we find that the optimal value for $\mathsf{baseAmpFactor}$ in practice is much greater than 2, which is the value implicitly suggested by all previous works.

(3) Optimizing the cache: SpeedCache. In the hierarchical ORAM solution, the top level (L_0 "the cache") needs to support completely oblivious accesses (compared to lower levels in the hierarchy which only need to be *distinct-query oblivious*). For this reason, L_0 is usually implemented as a simple read-all-to-read, append-to-write array. This is obviously oblivious, but its query complexity increases linearly with the size of the cache. In particular, if the cache stores t key-value pairs $((\llbracket x_1 \rrbracket, \llbracket y_1 \rrbracket), \ldots, (\llbracket x_t \rrbracket, \llbracket y_t \rrbracket))$, querying the cache is often implemented by sequentially checking whether the query, $\llbracket x \rrbracket$, is equal to $\llbracket x_i \rrbracket$ (costs $O(\log\log|x_i|)$ sequential rounds) and if so updating return value to $\llbracket y_i \rrbracket$. Unfortunately, this simple implementation has multiplicative depth t, meaning $\mathsf{numRoundsCache} = O(|L_0| \cdot \log\log|x_i|)$.

In Section 8, we outline a simple Cache protocol SpeedCache that allows us to query the cache in $\lceil \log\log|x_i| \rceil + 1$ rounds of communication (which is *independent of* $|L_0|$). Since our SpeedCache protocol has a round complexity that is independent of the cache size, we are somewhat free to increase the cache size, which has other benefits (e.g. reduce $\mathsf{numLevels}$ by $\log_2|L_0|$).

(4) Gadget implementations: Minimizing the round complexity of our SISO-PRF is crucial for the overall efficiency of our protocol, so we provide the first circuit file of the LowMC [1] block cipher within our custom MPC implementation. Our circuit file features a novel optimization we call "wire threading" that allows us to reduce the number of L1-cache misses during evaluation. See Appendix E for further details.

We adapt the Alibi reinsertion technique [18] for "caching the stash" to the distributed setting, and we provide the first implementation of Alibi. See Appendix F for further details.

6 ShufTable: reduce numRoundsOHTable

We present ShufTable, a novel, oblivious to build, asymptotically and practically efficient, oblivious to distinct-query, hash table (OHTable). Our chief goal with ShufTable is to minimize the round-complexity of a query, i.e., decreasing numRoundsOHTable without blowing up any other costs. Where possible, we also try improving the communication/-computation of [19]'s OHTable. We present two variations of ShufTable with identical build costs, where the one we implement has $O(\kappa)$ communication overhead over the other, but saves $O(\log \kappa) \approx 7$ rounds per query which results in a net efficiency gain in practice. In Section 9 we use the hierarchical solution and [18] to compile ShufTable into a round, bandwidth, and communication efficient DORAM. Overall, compared to [19]'s 45-rounds-per-query OHTable [9] (including necessary OSet sub-query), we reduce numRoundsOHTable ≈ 7 at, making the protocol much faster in practice.

In Section 6.1 we present the "Persistent shuffle protocol," a novel trick that enables $\Pi_{\text{ShufTable}}$, the new OHTable protocol we present in Section 6.2. Leveraging features of $\Pi_{\text{ShufTable}}$.Query, in Section 6.3 we show how to amortize the round cost of SISO-PRF evaluations across $L_1, \ldots, L_{\text{numLevels}}$.

6.1 Persistent shuffle Protocol

Like many DORAM protocols, our DORAM construction relies on an efficient oblivious shuffle, which allows players holding a secret shared lists, $[\![X]\!] = [\![X_1]\!], \ldots, [\![X_n]\!]$ to shuffle $[\![X]\!]$ under some random permutation $\pi \in S_n$ such that *nothing about* π *is learned by any player*.

As presented in [30], there is an extremely efficient, linear-communication (3,1)-oblivious shuffle which works as follows. The players P_1, P_2, P_3 reshare $[\![X]\!]$ to P_1, P_2, who shuffle their secret shares according to some random agreed upon permutation, and reshare the shuffled list to P_2, P_3, who shuffle and reshare to P_3, P_1 who shuffle and reshare to P_1, P_2, P_3. Since the composition the players permutations is known to no single player, the final permutation is oblivious. Since no two players collude, the secret sharing remains secure and thus no player learns X.

Unfortunately, at the end of the [30] protocol, information about the permutation, π, is not accessible to the players. In our Persistent shuffle, we augment the shuffling protocol to also output a secret sharing of π.

Our "Persistent shuffle" fulfills the functionality $\mathcal{F}_{\text{ABB}}$.ObliviousShuffle($\circ$, DistributeShuffle $= True$) where the players input $[\![X]\!]$ and receive as output $[\![\pi(X)]\!]$ and $[\![K]\!] = [\![K_1]\!], \ldots, [\![K_n]\!]$ s.t $\pi(i) = K_i$. It also guarantees that π is uniformly sampled from S_n and unknown to all players. Our novel "Persistent shuffle" is described in Figure 5 in Appendix B, and is used in step 4 of the $\Pi_{\text{ShufTable}}$.Build protocol (Figure 6).

[9] [19] and previous hierarchical solution DORAMs have round complexity that scales with N, κ while ours is constant

6.2 ShufTable construction

In this section, we describe and evaluate $\Pi_{\text{ShufTable}}$.Build and $\Pi_{\text{ShufTable}}$.Query, the components of our new Oblivious Hash Table ShufTable.

Parameters. $\Pi_{\text{ShufTable}}$ is parameterized by N, the number of elements in the DORAM, D, the size of each payload, κ, the computational security parameter. We use n to denote the number of elements stored in a specific level of the hierarchy. numDummies is the number of dummy elements stored in ShufTable. The variable stashSize is the minimum size of a stash in cuckoo hash table storing $O(n)$ elements such that the probability of a build failure is less than our statistical security parameter, σ.

On each query, we must be able to retrieve either (1) the element being queried or (2) a new dummy element that has never been queried (if the desired element is not in the OHTable). Hence, the number of queries to each ShufTable is bounded by $\min\{\text{numDummies}, n\}$. For this reason, we usually set numDummies $\approx n$. We recommend values for these parameters in Section 9.

Since our construction uses a Cuckoo hash table (CHT), it as also implicitly parameterized by c, the number of slots in CHT table, and t, the number of such tables. Instantiations of these variables are discussed in Section 9, and Section 10.

Input-Output behavior of $\Pi_{\text{ShufTable}}$.Build. The parties, P_1, P_2, P_3, input $E = \{([\![X_1]\!], [\![Y_1]\!]), \ldots, ([\![X_n]\!], [\![Y_n]\!])\}$ where $n \leq N$ to $\Pi_{\text{ShufTable}}$.Build. Protocol $\Pi_{\text{ShufTable}}$.Build, outputs secret shares of CHT held by P_2, P_3, $[\text{CHT}]^{(2,3)}$. The data structure, ShufTable, stores a subset of the elements sent to the build protocol $A \subset E$. $\Pi_{\text{ShufTable}}$.Build also outputs Stash $= E \setminus A$, with $|\text{Stash}| \leq$ stashSize. In the greater DORAM protocol (Section 9), Stash will be inserted into the cache, L_0. As long as $X_1, \ldots, X_n$ are distinct (as assured by the hierarchical solution), it is guaranteed that the parties learn nothing of $(X_1, Y_1), \ldots, (X_n, Y_n)$ or which elements belong to Stash.

Input-Output behavior of $\Pi_{\text{ShufTable}}$.Query. For the t'th query where $1 \leq t \leq$ numDummies, the players input $[\![x_t]\!]$ to $\Pi_{\text{ShufTable}}$.Query. $\Pi_{\text{ShufTable}}$.Query, outputs $[\![Y_j]\!]$ if $x_t = X_j$ for $(X_j, Y_j) \in A$ and $[\![\perp]\!]$ otherwise (and outputs $[\![\text{found}]\!]$ accordingly). The security guarantee is that for any sequence of *distinct* queries, $x_1, \ldots, x_t$, P_1, P_2, P_3 do not learn whether Y_j or $\perp$ was outputted, j, or Y_j.

Protocol $\Pi_{\text{ShufTable}}$ is presented in Figure 6 in Appendix C.

Succinct performance analysis. The cost of $\Pi_{\text{ShufTable}}$.Build is dominated by the cost of n SISO-PRF evaluations, which requires a total of $2304n$ bits of communication at $\kappa = 128$ and the computation of many XORs. Recall that [19] requires more than twice as many SISO-PRF evaluations ($2n +$ numDummies) for each Build,

and for each query. The rest of the steps involve orders of magnitude less communication. Yet, via multi-threading LowMC (Section E) we are able to evaluate a record-breaking 6,725,447 SISO-PRFs/s, so this cost is hardly felt. Until we significantly increase baseAmpFactor (sec. 7) we spend very little (less than 3%) of DORAM time in rebuild (see Section 10).

The time needed to evaluate $\Pi_{\text{ShufTable}}$.Query is driven by its round complexity, since we must sequentially invoke this protocol numLevels times during Π_{DORAM}.Query and send very little data at each step. Each step of the query requires equality checks on secret shared elements, which we execute using 3-party garbled circuits[10] . This makes the *computation* time of a query extremely low. In practice, each step of the query requires sending fewer than 2000 bits, independent of N.

6.3 Parallelizing sequential SISO-PRF evaluations

In the hierarchical ORAM construction, each query requires searching *every* level in the hierarchy, and the OHTable query at a given level requires evaluating a SISO-PRF.

In previous constructions (e.g. [19, 32]) these SISO-PRF were evaluated sequentially, because when performing the OHTable query for an index, x, at level i, the SISO-PRF input will be $[\![x]\!]$ or a dummy element depending on whether x was found in a smaller level of the hierarchy.

Since there are only two possible SISO-PRF inputs at each level, rather than evaluating the PRF sequentially, the players could evaluate *both* the "dummy query"

$$[\![a_i]\!] = \mathcal{F}_{\text{ABB}}.\text{PRFEval}([\![N+t_i]\!], [\![k]\!])$$

and the "real query"

$$[\![b_i]\!] = \mathcal{F}_{\text{ABB}}.\text{PRFEval}([\![X_{query}]\!], [\![k]\!])$$

for all $i \in \{1, \ldots, \mathsf{numLevels}\}$ in parallel before querying L_1, then multiplex the result using MPC (which costs only a single round) before evaluating L_i.Query.

This trick will reduce the *round* complexity, but requires *doubling* the number of SISO-PRF calls per query. Since round-complexity is often the bottleneck, this will likely improve practical performance even if it increases overall communication.

In our protocol, however, we can leverage the design of ShufTable to parallelize the SISO-PRF calls *without* increasing communication.

The crucial observation is that we have designed ShufTable to require only $[\![b_i]\!]$, regardless of found. That is, since previous OHTables stored and retrieved their dummy elements from some data structure, obtaining the index of each dummy from a_i was needed. We observe that it is not necessary, using the $(3,1)$ garbled circuits of [33] to "just-in-time" detect if q_i is stored in the table and output a dummy index if necessary ($\Pi_{\text{ShufTable}}$.Query, step 4). Thus, if found_{i-1}, for ShufTable it is sufficient to set $q_i = r$ to be some uniformly random κ-bit value r ($\Pi_{\text{ShufTable}}$.Query, step 1), which, except from with negligible probability, will not be stored in the table and will yield the desired dummy-output. This "just-in-time" trick is only possible due to the Persistent shuffle trick we present in Section 6.1.

Thus to parallelize the SISO-PRF, we evaluate $[\![b_i]\!] = \mathcal{F}_{\text{ABB}}.\text{PRFEval}([\![X_{query}]\!], [\![k]\!])$ for all $i \in \{1, \ldots, \mathsf{numLevels}\}$ in parallel before querying L_1. Hence we can parallelize the SISO-PRF evaluations across the table without increasing the total number of SISO-PRF calls.

This optimization reduces numRoundsDORAM by $(\mathsf{numLevels} - 1) \cdot \mathsf{numRoundsPRFEval}$. To give a sense of these variables, in previous constructions $\mathsf{numLevels} = \log_2 N$ and $\mathsf{numRoundsPRFEval} > 20$, saving ≈ 500 rounds at $N = 2^{25}$. Due to further optimizations, we have $\mathsf{numLevels} \approx 5$ and $\mathsf{numRoundsPRFEval} = 9$, saving us ≈ 45 rounds per DORAM query.

7 Reducing the depth of the hierarchy

We introduce a novel adjustable parameter to the hierarchical solution, the "base amplification factor" denoted baseAmpFactor. Increasing baseAmpFactor decreases numLevels by a factor of $\log(\mathsf{baseAmpFactor})$ and increases the amortized communication per query by a factor of $\mathsf{baseAmpFactor}/\log(\mathsf{baseAmpFactor})$. While in theory this might not sound impressive, increasing baseAmpFactor dramatically improves practical performance because latency is significantly more time-expensive than bandwidth (Section 10). For instance, for GigaDORAM we found that increasing baseAmpFactor with N to maintain that $\mathsf{numLevels} \approx 5$ and $|L_0| \approx 2^{10}$ yielded the best performance. For $N = 2^{30}$, this means setting $\mathsf{baseAmpFactor} = 16$, which is much larger than all previous protocols, which implicitly set $\mathsf{baseAmpFactor} = 2$.

In most hierarchical ORAM solutions, each level is twice as large as the level above it, i.e., $|L_{i+1}| = 2|L_i|$. Since we must have $|L_{\mathsf{numLevels}}| = O(N)$ and generally $|L_0|$ is a small constant, this means that $\mathsf{numLevels} = \log N$. Coming from this view, at a high level, our optimization sets $|L_{i+1}| \approx \mathsf{baseAmpFactor} \cdot |L_i|$, reducing numLevels by $\log_2 \mathsf{baseAmpFactor}$.

In most hierarchical ORAM schemes (where $\mathsf{baseAmpFactor} = 2$), when levels $L_0, \ldots, L_i$ are full, they are reshuffled and rebuilt into L_{i+1} Since

$$\sum_{j=0}^{i} |L_j| = \sum_{j=0}^{i} 2^j |L_0| = (2^{i+1} - 1)|L_0|$$

[10] 3-party garbled circuits are significantly more bandwidth efficient than BMR [6].

this rebuild schedule works nicely when $\mathsf{baseAmpFactor} = 2$. Another way to describe this schedule is that L_{i+1} is rebuilt (with $(2^{i+1}-1)\cdot|L_0|$ elements) after the $t = (2^{i+1}-1)\cdot|L_0|$ queries into the ORAM.

In our generalization, we also rebuild levels when they are full, but

$$\mathsf{baseAmpFactor}^{i+1}\cdot|L_0| \gg \sum_{j=0}^{i}\mathsf{baseAmpFactor}^{j}|L_0|$$

so we adjust the rebuild schedule slightly to accommodate "partial rebuilds." We formalize this procedure in protocol Π_{DORAM}.Rebuild presented in Appendix G, Figure 8.

Since rebuilding level $i+1$ costs at most $O\left(\kappa\cdot|L_0|\cdot\mathsf{baseAmpFactor}^{i+1}\right)$ communication and computation, the total amortized (re)build cost of the DORAM is at most

$$\sum_{i=1}^{\frac{\log(N)}{\log(\mathsf{baseAmpFactor})}}\frac{O\left(\kappa\cdot|L_0|\cdot\mathsf{baseAmpFactor}^{i+1}\right)}{|L_0|\cdot\mathsf{baseAmpFactor}}$$
$$= O\left(\frac{\kappa\cdot\mathsf{baseAmpFactor}\cdot\log(N)}{\log(\mathsf{baseAmpFactor})}\right)$$

Since the original amortized (re)build cost of the DORAM is $O(\kappa\log N)$, the overhead of computation and communication overhead of increasing $\mathsf{baseAmpFactor}$ is $O(\mathsf{baseAmpFactor}/\log(\mathsf{baseAmpFactor}))$ rounds.

8 SpeedCache: Larger, optimized cache

Most hierarchical (D)ORAMs use a constant-sized "cache," i.e., $|L_0| = O(1)$. Since each ORAM query performs a linear scan over the cache (implying a linear communication, computation, and number of rounds), a large cache can significantly hurt both asymptotics and practical performance. In our protocol, we use the Alibi reinsertion technique to "cache-the-stash" from Cuckoo hash tables at each level of the ORAM hierarchy. This means that we must have a moderately large cache size, otherwise reinserting the stashes would fill the stash and trigger an infinite chain of table rebuilds.

Thus we increase the cache size to $|L_0| = \Omega(\mathsf{stashSize})$. This has the additional advantage of eliminating "small" levels from the ORAM hierarchy, i.e., L_i s.t $|L_i| < |L_0|$.[11]

In order to facilitate our new, larger cache without hurting query complexity, we outline a new data structure, SpeedCache, that can be queried using $O((D+\log N)\cdot|L_0|)$ communication and only requires $\lceil\log\log N\rceil+1$ rounds of communication (that is, *independent of the number of elements in the cache* – only dependent on the size of the address space).

[11] Because cuckoo hash tables have higher build failure probability when the table size is small, many hierarchical (D)ORAMs had two types of tables, one for the "small" levels and one for the "large" levels (e.g. [19, 32]). Because our L_0 is sufficiently large, we do not incur this complexity.

Our SpeedCache protocol is both round-efficient *and* communication efficient, and we implement it in the 3-party MPC framework of [2]. We present the protocol, $\Pi_{\text{SpeedCache}}$, in Figure 7 in Appendix D.

Experimentally we find that it is optimal to maintain $|L_0| \approx 2^{10}$ a constant, rather than scaling $|L_0|$ with N. Note that $2^{10} > \log(N)$ for any conceivable value of N. Concretely, setting $|L_0| \approx 2^{10}$ as we do in practice allows us to reduce $\mathsf{numLevels}$ by $\approx 10/\log_2(\mathsf{baseAmpFactor})$.

An alternative approach would be to implement the trivial linear-scan cache (see Appendix D) via garbled circuits in a constant number of rounds. When $|L_0| \approx 2^{10}$ we estimate this would make L_0.Query up to $100\times$ slower than our implementation of SpeedCache.

9 Implementation

DORAM protocol. The full DORAM protocol is presented in Figure 8 in Appendix G. With our optimizations the final round complexity of Π_{DORAM}.Query is

$$\begin{aligned}&\mathsf{numRoundsDORAM} =\\ &\quad\mathsf{numLevels}\cdot\mathsf{numRoundsOHTable}\\ &\quad+\max(\mathsf{numRoundsCache},\mathsf{numRoundsPRFEval})+1\\ &\quad=\mathsf{numLevels}\cdot 5+\max(\log\log(N)+2,9)+1\\ &\quad\approx 5\cdot 5+9+1=35 \qquad (\text{when } N=2^{31})\end{aligned}$$

The term $\max(\log\log(N)+2,9)$ occurs because L_0.Query ($\log\log(N)+2$ rounds) and the SISO-PRF pre-evaluations (9 rounds – see Section 6.3, Appendix E) are evaluatable in parallel. Hence, compared to the over 2500-round-per-query [19] construction we draw inspiration from, GigaDORAM has a $45\times$ reduction in round complexity. Our overall communication complexity is comparable to that of [19, 32] and previous hierarchical DORAMs.

Open-source Contributions. We provide several open source contributions.

1. Custom C++, high performance, open source, implementation of GigaDORAM. Counting only source files written by us, our implementation is over 9000 lines.

2. Python and Javascript implementations of GigaDORAM clients for the client-server setting.

3. From-scratch, competitive, multi-threaded implementation of [2]. The benchmarks provided in [2] were done using a proprietary implementation, which is not widely available, and to the best of our knowledge, has not been implemented outside of giant MPC frameworks.

4. an alternative implementation of the ABY3's 3-party garbled circuit protocol [33]. Our 3-party garbling is built on the fast garbling function provided by EMP toolkit [47]. Although the ABY3 code is available [42],

we only need the 3-party garbled circuit functionality and it was challenging to integrate ABY3 with the rest of our code.

5. The first circuit files of the LowMC [1] cipher. Our circuit files are written in the popular Bristol Fashion format and thus they are suitable for execution by EMP toolkit and other MPC frameworks. We provide both a faster "wire threaded" version and a traditional version. We also provide the flexible Python scripts we use to produce a variety of the circuit files used in our protocol.

Additionally, Once the paper is deanonymized we will integrate our codebase with the popular, actively maintained, open source, EMP-toolkit [47]. We hope this will enable people to use and build more complex applications on top of our DORAM both in client-server and RAM-MPC setting. We are most excited to use DORAM to add a random access operations to EMP's standard MPC framework, which will enable non-expert users to build RAM-MPC applications which were previously impossible (due to computation in the circuit model).

Multithreading. We use the `std:thread` library to multithread the SISO-PRF evaluations required to build a level. Using 30 threads on our evaluation hardware, we were able to achieve 6,725,447 SISO-PRFs per second, over 5× the previous record of [2] of 1,300,000!

10 Evaluation

In this section we evaluate the practical performance of GigaDORAM by comparing it to previous DORAM implementations (Figure 3). Our benchmarks go to great lengths to ensure that all DORAM are tested on comparable hardware. Our benchmarks reveal that GigaDORAM outperforms all previous DORAMs for $N \geq 2^{12}$. While for small memories ($\log N \in [12,20]$) GigaDORAM improves on the competition by a nice factor of $4-12\times$, for medium memories ($\log N \in (20,25]$) GigaDORAM improves on the competition by a significant factor $20-150\times$, and for large memories ($\log N \in (26,29]$) GigaDORAM improves on the competition by a *tremendous* factor of $300-2000\times$, for *very* large memories ($30 \leq \log N$), GigaDORAM improves on the competition by a ***tremendous*** factor of over $7000\times$.

Testing environment. To make our results easily reproducible, we test on AWS. AWS solution engineers recommended we use *bare metal* instances[12] for a "very high probability" that our machines will be physically seperate. Bare metal instances range from 192 to 4096 GiB memory, from 48 to 192 vCPUs, and 25-100 Gigabit bandwidth. For our testing we use relatively-lightwight c5.metal instances which have 96vCPUs, 192GiB memory, 25Gb/s bandwidth. To the best of our knowledge, our experiments with GigaDORAM did not exceed 1/3 utilization. We deploy three such Ubuntu 22.04 LTS (HVM) SSD Volume Type, 64-bit (x86) CPU, c5.metal instances in cluster placement at region `us-west-2`. Via `ping -c 20` we mesure latency of `min/avg/max/mdev = 0.229/0.236/0.244/0.003 ms` between a pair of cluster-placed benchmarking servers. To note, we have observed latency fluctuating according to usage, although AWS does not make such policies public. We compile our code via `gcc` with the `-pthread -Wall -funroll-loops -Wno-ignored-attributes -Wno-unused-result -march=native -maes -mrdseed -std=c++11 -O3` flags.

Testing environment for other works. To compare fairly with previous works, we *re-benchmark all previous DORAMs with* existing implementations (with the exception of proprietary [26]) *on the same hardware setup we test GigaDORAM*.

We benchmark DuORAM [45] via their remarkably convenient Docker setup on a single c5.metal machine. We use the `set-networking.sh` script they provide to set .229ms latency and 25Gbit bandwidth simulated network between their Docker containers. We do not restrict the number of cores their process can use, and indeed, using `htop` we can see DuORAM use all 96 vCPUs during their preprocessing stage. Due to a precarious memory leak DuORAM's query/second figures are somewhat higher than they should be for $N > 2^{20}$ as they do not include preprocessing costs. [13]

We benchmark FLORAM, FLORAM-CPRG, circuit ORAM, and square root ORAM [16, 46, 53] via the obliv-c [52] based setup given by [16]. [14] We benchmark the above 2-party constructions between two cluster-placed c5.metal machines in the `us-west-2` region. For backwards compatibility with obliv-c, it was necessary to run Ubuntu 18.04.6 LTS. We do not artificially restrict the network via the `tc` command as was done in [16].

Bingsheng Zhang kindly benchmarked the proprietary PFE-DORAM [26] on a comparable network to ours. Zhang executed the protocol via separate processes on the same Intel(R) Core i7 8700 CPU 3.2 GHz, 6 CPUs, 32 GB Memory, 1TB SSD machine running Ubuntu 18.04.2 LTS. Bandwidth between the processes was not limited and latency was restricted to 0.05ms.

Discussion of testing environment. The main difference between our experiments and those conducted by prior work are the network settings. We use good (yet not best available to consumers) inter-server network while previous works *artificially* restrict their servers' connection (e.g. [16, 26, 45]). For instance, DuORAM [45] artificially restricted the network between their servers (i.e. Docker containers on the same machine) to a bandwidth of 100Mb/s (250× lower than us)

[12] https://aws.amazon.com/about-aws/whats-new/2021/11/amazon-ec2-bare-metal-instances/

[13] We do not suggest the leak is due to to DuORAM's code and may entirely be our fault. We are satisfied with reporting an upper bound on DuORAM's query/sec for $N > 2^{20}$.

[14] See https://gitlab.com/neucrypt/floram/

and latency of 30ms (1000× higher than us). While it is quite interesting to see DORAMs performing well when the network amongst the servers conditions are so poor and believe this direction of research is very important, we do not think that these settings represent the practical common case. That is because MPC is not currently practical in these network settings, we show a performance boost sufficiently large to justify co-locating servers, and superior network conditions are ubiquitously available even for distant servers. [15]

The experiments. We decided to measure only writes in our experiments because "reads" vary across works (e.g. DuORAM considers the notion of "dependent" and "independent" reads) and for all constructions, writes are at least as expensive as reads. For each construction at each N we averaged performance across 512 to 1024 writes, depending on the number of queries we managed to measure in reasonable time. GigaDORAM consistently high query/sec allowed us to average over 100,000 queries. All experiments were on $D = 64$ bit payloads as in DuORAM's testing.

Discussion of results (Figure 3). Figure 3 shows that GigaDORAM outperforms all other DORAMs for $2^{12} \leq N \leq 2^{31}$. GigaDORAM perform $600-900$ queries/sec for all measured N, with oscillations which depend on the size of the hierarchical cache L_0. [16] In the plot, one can see the performance of FSS-based schemes decrease linearly with N as predicted by the linear compute asymptotics. All FSS-based schemes vanish below 10 queries per second at 2^{24} and below 1 query per second at 2^{28}. Circuit DORAM and Square Root DORAM, trailblazing works in practical DORAM, use techniques which are closer in vain to "simulate the client under MPC DORAMs" (described in Section 1.2) and hence preform worse than all other DORAMs.

Further scaling of GigaDORAM As Figure 3 shows, we have not reached the scaling the limit of our DORAM. We don't know what those limits are, and are not sure what the bottlenecks may be. Unfortunately, our implementation (but not our construction) assumes DORAM addresses are 32 bits long, and thus we were not able to test for $N > 2^{31}$. We leave this interesting exploration to future work.

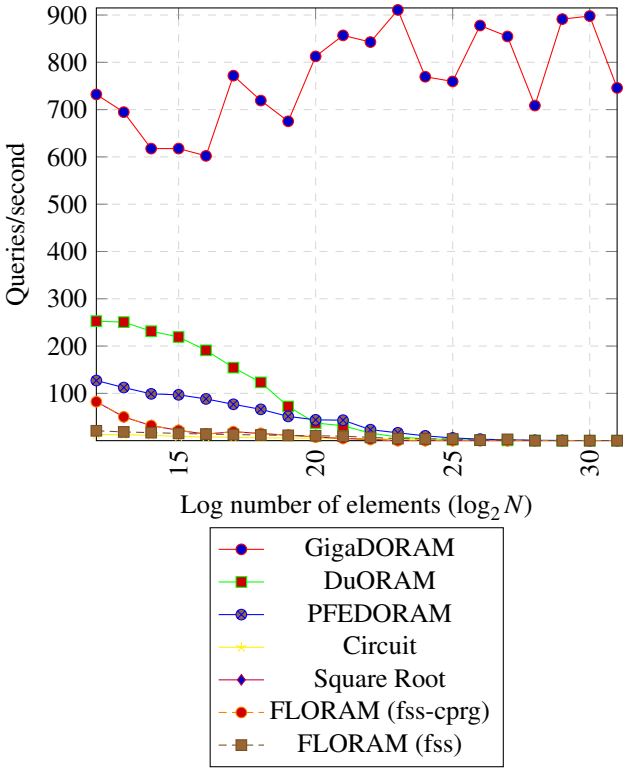

Figure 3: Number of queries/second vs. log number of elements (N). Each query is a write of random data to a random index. Preprocessing and initialization costs are accounted for. The number of queries per second is an estimate of (at least) hundreds of queries. All constructions were ran on comparable hardware..

11 Related work

Although Oblivious RAM has been widely studied, only a handful of different DORAM protocols have been implemented.

[24] uses garbled circuits to implement the [43] ORAM protocol in the two-party setting. SCORAM [48] uses the ObliVM [49] framework to implement a 2-party DORAM protocol. Using Obliv-C [52], the square-root DORAM [53], the circuit DORAM [46]. In today's standards, these trailblazing constructions are not very efficient, often falling below 10 queries per second already at 2^{17}. [16] introduced Function-Secret-Sharing (FSS) [7, 20] as a technique to build practically efficient, computation bound DORAMs. [16] was later improved on by [26, 45] which further refined FSS-based DORAMs.

12 Conclusion

In this work we introduce GigaDORAM, the most efficient and scalable DORAM construction to date. At $N = 2^{31}$, our DORAM can perform over 700 queries per second, making GigaDORAM over 2500× faster than prior work. We give a custom C++, open source implementation of GigaDORAM which we intend to contribute to EMP-toolkit once the paper is deanonymized. We hope GigaDORAM will enable the first somewhat practical RAM-MPC applications and open a new

[15] We measured 20ms ping from `us-east-1` to `us-west-2` and https://aws.amazon.com/blogs/aws/the-floodgates-are-open-increased-network-bandwidth-for-ec2-instances/ suggest servers in different AWS regions should enjoy 5Gbit connections.

[16] The "oscillations" seen in our performance are due to the fact $\log_2 |L_0| = N - \mathsf{baseAmpFactor} \cdot (\mathsf{numLevels} - 1)$ and the fact that $\mathsf{baseAmpFactor}$ is a power of 2, and thus for certain values of N we cannot reasonably set $|L_0|$ as small as we'd like.

realm of possibilities for privacy-preserving cloud data-stores.

References

[1] ALBRECHT, M. R., RECHBERGER, C., SCHNEIDER, T., TIESSEN, T., AND ZOHNER, M. Ciphers for MPC and FHE. In *EUROCRYPT* (2015), Springer, pp. 430–454.

[2] ARAKI, T., FURAKAWA, J., LINDELL, Y., NOF, A., AND OHARA, K. High-throughput semi-honest secure three-party computation with an honest majority. In *CCS* (2016).

[3] ASHAROV, G., KOMARGODSKI, I., LIN, W.-K., NAYAK, K., PESERICO, E., AND SHI, E. OptORAMa: Optimal oblivious RAM. In *EUROCRYPT* (2020).

[4] ASHAROV, G., KOMARGODSKI, I., LIN, W.-K., AND SHI, E. Oblivious RAM with worst-case logarithmic overhead. In *CRYPTO* (2021), Springer, pp. 610–640.

[5] BANIK, S., BAROOTI, K., VAUDENAY, S., AND YAN, H. New attacks on lowmc instances with a single plaintext/ciphertext pair. Cryptology ePrint Archive, Paper 2021/1345, 2021. https://eprint.iacr.org/2021/1345.

[6] BEAVER, D., MICALI, S., AND ROGAWAY, P. The round complexity of secure protocols. In *Proceedings of the twenty-second annual ACM symposium on Theory of computing* (1990), pp. 503–513.

[7] BOYLE, E., GILBOA, N., AND ISHAI, Y. Function secret sharing. In *EUROCRYPT* (2015).

[8] BUNN, P., KATZ, J., KUSHILEVITZ, E., AND OSTROVSKY, R. Efficient 3-party distributed ORAM. In *SCN* (2020).

[9] CECCHETTI, E., ZHANG, F., JI, Y., KOSBA, A., JUELS, A., AND SHI, E. Solidus: Confidential distributed ledger transactions via pvorm. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (2017), pp. 701–717.

[10] CHASE, M., DERLER, D., GOLDFEDER, S., ORLANDI, C., RAMACHER, S., RECHBERGER, C., SLAMANIG, D., AND ZAVERUCHA, G. Post-quantum zero-knowledge and signatures from symmetric-key primitives. Cryptology ePrint Archive, Paper 2017/279, 2017. https://eprint.iacr.org/2017/279.

[11] CHAUM, D., CRÉPEAU, C., AND DAMGÅRD, I. Multiparty Unconditionally Secure Protocols. In *STOC* (1988).

[12] CONNELL, G. Technology deep dive: Building a faster ORAM layer for enclaves. https://signal.org/blog/building-faster-oram/, August 2022.

[13] CORRIGAN-GIBBS, H., BONEH, D., AND MAZIÈRES, D. Riposte: An anonymous messaging system handling millions of users. In *2015 IEEE Symposium on Security and Privacy* (2015), IEEE, pp. 321–338.

[14] CRAMER, R., DAMGÅRD, I., AND ISHAI, Y. Share conversion, pseudorandom secret-sharing and applications to secure computation. In *TCC* (2005).

[15] DINUR, I., LIU, Y., MEIER, W., AND WANG, Q. Optimized interpolation attacks on lowmc. Cryptology ePrint Archive, Paper 2015/418, 2015. https://eprint.iacr.org/2015/418.

[16] DOERNER, J., AND SHELAT, A. Scaling ORAM for secure computation. In *CCS* (2017).

[17] FALK, B. H., LU, S., AND OSTROVSKY, R. Durasift: A robust, decentralized, encrypted database supporting private searches with complex policy controls. In *Proceedings of the 18th ACM Workshop on Privacy in the Electronic Society* (2019), pp. 26–36.

[18] FALK, B. H., NOBLE, D., AND OSTROVSKY, R. Alibi: A flaw in cuckoo-hashing based hierarchical ORAM schemes and a solution. In *EUROCRYPT* (2021), Springer, pp. 338–369.

[19] FALK, B. H., NOBLE, D., AND OSTROVSKY, R. 3-party distributed ORAM from oblivious set membership. In *International Conference on Security and Cryptography for Networks* (2022), Springer, pp. 437–461.

[20] GILBOA, N., AND ISHAI, Y. Distributed point functions and their applications. In *EUROCRYPT* (2014).

[21] GOLDREICH, O., MICALI, S., AND WIGDERSON, A. How to play any mental game. In *STOC* (1987).

[22] GOLDREICH, O., AND OSTROVSKY, R. Software protection and simulation on oblivious RAMs. *JACM 43*, 3 (1996).

[23] GOODRICH, M. T., MITZENMACHER, M., OHRIMENKO, O., AND TAMASSIA, R. Privacy-preserving group data access via stateless oblivious RAM simulation. In *SODA* (2012).

[24] GORDON, S. D., KATZ, J., KOLESNIKOV, V., KRELL, F., MALKIN, T., RAYKOVA, M., AND VAHLIS, Y. Secure two-party computation in sublinear (amortized) time. In *CCS* (2012).

[25] GRASSI, L., RECHBERGER, C., ROTARU, D., SCHOLL, P., AND SMART, N. P. MPC-friendly symmetric key primitives. In *CCS* (2016), pp. 430–443.

[26] JI, K., ZHANG, B., LU, T., AND REN, K. Multi-party private function evaluation for RAM. Cryptology ePrint Archive, Paper 2022/939, 2022. https://eprint.iacr.org/2022/939.

[27] KIRSCH, A., MITZENMACHER, M., AND WIEDER, U. More robust hashing: Cuckoo hashing with a stash. *SIAM Journal on Computing* (2009).

[28] KUSHILEVITZ, E., LU, S., AND OSTROVSKY, R. On the (in) security of hash-based oblivious RAM and a new balancing scheme. In *SODA* (2012).

[29] LARSEN, K. G., AND NIELSEN, J. B. Yes, there is an oblivious RAM lower bound! In *CRYPTO* (2018).

[30] LAUR, S., WILLEMSON, J., AND ZHANG, B. Round-efficient oblivious database manipulation. In *ISC* (2011).

[31] LIU, F., ISOBE, T., AND MEIER, W. Cryptanalysis of full lowmc and lowmc-m with algebraic techniques. Cryptology ePrint Archive, Paper 2020/1034, 2020. https://eprint.iacr.org/2020/1034.

[32] LU, S., AND OSTROVSKY, R. Distributed oblivious RAM for secure two-party computation. In *TCC* (2013).

[33] MOHASSEL, P., AND RINDAL, P. ABY3: A mixed protocol framework for machine learning. In *CCS* (2018), pp. 35–52.

[34] NAVEED, M. The fallacy of composition of oblivious ram and searchable encryption. IACR ePrint 2015/688, 2015.

[35] NIST. Post-quantum cryptography PQC: Round 3 submissions. https://csrc.nist.gov/Projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-3-submissions, 2021.

[36] NOBLE, D. An intimate analysis of cuckoo hashing with a stash. IACR ePrint 2021/447, 2021.

[37] OSTROVSKY, R. Efficient computation on oblivious RAMs. In *STOC* (1990).

[38] OSTROVSKY, R. *Software Protection and Simulation On Oblivious RAMs*. PhD thesis, Massachusetts Institute of Technology, 1992.

[39] OSTROVSKY, R., AND SHOUP, V. Private information storage. In *STOC* (1997), vol. 97.

[40] PATEL, S., PERSIANO, G., RAYKOVA, M., AND YEO, K. PanORAMa: Oblivious RAM with logarithmic overhead. In *FOCS* (2018).

[41] PINKAS, B., SCHNEIDER, T., SEGEV, G., AND ZOHNER, M. Phasing: Private set intersection using permutation-based hashing. Cryptology ePrint Archive, Paper 2015/634, 2015. https://eprint.iacr.org/2015/634.

[42] RINDAL, P. The ABY3 Framework for Machine Learning and Database Operations. https://github.com/ladnir/aby3.

[43] SHI, E., CHAN, T.-H. H., STEFANOV, E., AND LI, M. Oblivious RAM with $O((\log N)^3)$ worst-case cost. In *ASIACRYPT* (2011).

[44] STEFANOV, E., VAN DIJK, M., SHI, E., FLETCHER, C., REN, L., YU, X., AND DEVADAS, S. Path ORAM: an extremely simple oblivious RAM protocol. In *CCS* (2013).

[45] VADAPALLI, A., HENRY, R., AND GOLDBERG, I. Duoram: A bandwidth-efficient distributed ORAM for 2- and 3-party computation. Cryptology ePrint Archive, Paper 2022/1747, 2022. https://eprint.iacr.org/2022/1747.

[46] WANG, X., CHAN, H., AND SHI, E. Circuit ORAM: On tightness of the Goldreich-Ostrovsky lower bound. In *CCS* (2015).

[47] WANG, X., MALOZEMOFF, A. J., AND KATZ, J. EMP-toolkit: Efficient multiparty computation toolkit. https://github.com/emp-toolkit/emp-sh2pc, 2016.

[48] WANG, X. S., HUANG, Y., CHAN, T.-H. H., SHELAT, A., AND SHI, E. SCORAM: oblivious RAM for secure computation. In *CCS* (2014).

[49] WANG, X. S., LIU, C., NAYAK, K., HUANG, Y., AND SHI, E. ObliVM: A programming framework for secure computation. In *S & P* (2015).

[50] YAO, A. Protocols for secure computations (extended abstract). In *FOCS* (1982).

[51] YAO, A. How to generate and exchange secrets. In *FOCS* (1986).

[52] ZAHUR, S., AND EVANS, D. Obliv-C: A language for extensible data-oblivious computation. IACR ePrint 2015/1153, 2015.

[53] ZAHUR, S., WANG, X., RAYKOVA, M., GASCÓN, A., DOERNER, J., EVANS, D., AND KATZ, J. Revisiting square-root ORAM: efficient random access in multi-party computation. In *S & P* (2016).

A ABB Functionalities

The basic ABB operations we use are described in Figure 4.

B Reverse Shuffle

Performance Analysis. The protocol is 4 rounds of communication and in each rounds we send $n(|X_i| + \log n)$ bits for a total of less than $8n(|X_i| + \log n)$ communication. To note, evaluating a shuffle on n elements is 6 times less bandwidth than evaluating a SISO-PRF on those elements, so this cost is not much-felt. The compute is negligible.

C The ShufTable protocol: $\Pi_{\text{ShufTable}}$

We give the ShufTable protocol, $\Pi_{\text{ShufTable}}$, in Figure 6

Obliviousness. Below we outline why $\Pi_{\text{ShufTable}}$ is distinct-query oblivious.

We begin by showing that the view of each player during $\Pi_{\text{ShufTable}}$.Build is independent of the input data X, Y. During $\Pi_{\text{ShufTable}}$.Build, P_2, P_3 only receive secret shares and operate on those secret shares using MPC. Since secret shares are sampled from a (individually) uniformly random distribution, P_2, P_3's views are independent of underlying data X, Y.

During $\Pi_{\text{ShufTable}}$.Build, other than operating on secret shares, P_1 receives the list $\hat{Q}$ in the clear. Yet, since Q is computationally independent of X and $\hat{Q}$ is a uniformly random shuffle of Q which P_1 does not know, P_1 cannot decipher *any* information about X from seeing Q. Moreover, when deciding which elements to place in Stash (by deciding on which indexes to place there), P_1 cannot tell which elements from (X_i, Y_i) he is placing in stash (except that they aren't dummies since $\hat{Q}_i \neq \perp$). Thus, P_1's view is independent of X, Y.

Next we show that the view of each player is independent of the query X_t, the output, or X, Y previously stored. During $\Pi_{\text{ShufTable}}$.Query, other than operating on secret shares, P_2, P_3 learn q in the clear. q is either a random or a pseudorandom element which is thus computationally independent from P_2, P_3's view (because P_2, P_3 did not see $\hat{Q}$ which P_1 used to build $\mathsf{CHT} \cup \mathsf{Stash}$). Thus, P_2, P_3 do not learn anything by seeing q. Additionally, P_2, P_3 learn l, an index into the $\wedge$-shuffled list which depends on X_t's presence at the level (see step 4 of $\Pi_{\text{ShufTable}}$.Query). Yet, since l is entirely determined by $\pi_\wedge$ which P_2, P_3 don't know, and moreover, since unlike P_1 they do not know which indexes of $\hat{Y}$ correspond to dummies, l is sampled from a distribution indistinguishable from random relative to the individual view of P_2, P_3.

Finally, during $\Pi_{\text{ShufTable}}$.Query, other than operating on secret shares, P_1 learns l in the clear. Yet, although P_1 knows which elements from $\hat{Y}$ are dummies, due to an additional oblivious shuffle, he does not know which elements of $\hat{\hat{Y}}$ are dummies (or were stashed, or were stored in the table, etc). Hence l is independent of P_1's view.

Thus, on distinct queries and when $i \neq j \implies X_i \neq X_j$, P_1, P_2, and P_3's views are independent of the data stored and queried, and hence ShufTable is distinct query oblivious.

D The SpeedCache protocol: $\Pi_{\text{SpeedCache}}$

The naïve cache protocol works as follows

$r \leftarrow \perp$
for $i = 1, \ldots, t$ **do**
  if $x = x_i$ **then**
    $r \leftarrow y_i$
  end if
end for
Return: r

Unfortunately, this code has multiplicative depth, t, thus with implementing this under the MPC of [2] leads to a protocol with $\mathsf{numRoundsCache} = |L_0|$. It is possible to implement this with a garbled-circuit-based approach, (e.g. 3-party garbled circuits [33] or BMR [6]). This results in a constant-round MPC protocol, but the communication complexity is extremely large, and the resulting protocol is inefficient in practice.

Below, we outline our new SpeedCache protocol, $\Pi_{\text{SpeedCache}}$, that effectively parallelizes the equality tests of the naïve protocol. Since $\Pi_{\text{SpeedCache}}$ is naturally a low-

- $[x]^{(i,j)} = \mathcal{F}_{\text{ABB}}.\text{InputTo2Sharing}(x, P_i, P_j)$. A single player shares creates an additive sharing of a secret, x, among participants i and j.
- $[x]^{(i,j)} = \mathcal{F}_{\text{ABB}}.\text{ReshareReplicatedTo2Sharing}([\![x]\!], \{P_i, P_j\})$. Convert a 3-party CNF sharing of a secret, x, to a two-party DNF sharing of the same secret, x, held by participants i and j.
- $[x]^{(i',j')} = \mathcal{F}_{\text{ABB}}.\text{Reshare2To2Sharing}([x]^{(i,j)}, \{P_{i'}, P_{j'}\})$. Convert a two-party DNF sharing of a secret, x, held by participants i and j, to a two-party DNF sharing of the same secret, x, held by participants i' and j'.
- $[\![x]\!] = \mathcal{F}_{\text{ABB}}.\text{Reshare2SharingToReplicated}([x]^{(i,j)})$. Convert a two-party DNF sharing of a secret, x, held by participants i and j to a three-party CNF sharing of the same secret, x.
- $[\![k]\!] = \mathcal{F}_{\text{ABB}}.\text{RandomElement}(\kappa)$. Generate a three-party CNF sharing of a uniformly random field element (whose value is unknown to the participants).
- $x = \mathcal{F}_{\text{ABB}}.\text{RevealTo}(P_i, [\![x]\!])$. Reveal a secret-shared value, $[\![x]\!]$, to participant P_i.

We also abstract away a few more sophisticated operations:

- $[\![y]\!] = \mathcal{F}_{\text{ABB}}.\text{PRFEval}([\![x]\!], [\![k]\!])$. Evaluate a SISO-PRF with secret-shared key, $[\![k]\!]$, on secret-shared input, $[\![x]\!]$, to obtain a secret-shared output, $[\![y]\!]$. In our instantation, we instantiate the $\mathcal{F}_{\text{ABB}}.\text{PRFEval}(\cdot,\cdot)$ functionality by evaluating the LowMC block cipher [1] under MPC. Specifically, in our custom implementation of a (3,1)-MPC protocol (based on [2]).
- $[\![\mathsf{QueryCircuit}(x_1,\ldots,x_l)]\!] = \mathcal{F}_{\text{ABB}}.\text{EvalCircuit(3,1)GC}(\mathsf{QueryCircuit}, \text{Inputs} = ([\![x_1]\!],\ldots,[\![x_l]\!]))$. Evaluate a 3-party garbled circuit on secret-shared inputs $[\![x_1]\!],\ldots,[\![x_l]\!]$, and returns a *sharing* of the output of the circuit computation. We use a custom implementation of the 3-party Garbled Circuit protocol outlined in ABY3 [33].
- $[\![Y]\!] = \mathcal{F}_{\text{ABB}}.\text{ObliviousShuffle}([\![X]\!])$. This functionality implements a linear-communication, three-party shuffle of secret shared values [30]. So $Y = \pi(X)$ for some random permutation, unknown to the participants. We also use a modified protocol to output a sharing of the permutation as well. $[\![Y]\!], [\![L]\!] = \mathcal{F}_{\text{ABB}}.\text{ObliviousShuffle}([\![X]\!], \text{DistributeShuffle} = \mathit{True})$. In this setting, $Y = \pi(X)$ as before, and $L = \pi^{-1}(1,2,\ldots,n)$. We describe how to implement this novel "Persistent shuffle" in Section 6.1.

Figure 4: The ABB functionalites used in our DORAM protocol.

Setup: Each pair of players P_i, P_j agree on a random permutation $\pi_{\{i,j\}} \in S_n$. The players also generate a sharing $[\![L]\!] = [\![1]\!], \ldots, [\![n]\!]$.
Protocol:

1. The players reshare to the first shufflers, calling

$$[X]^{(1,2)} = \mathcal{F}_{\text{ABB}}.\text{ReshareReplicatedTo2Sharing}([\![X]\!], \{P_1, P_2\})$$

and

$$[L]^{(3,1)} = \mathcal{F}_{\text{ABB}}.\text{ReshareReplicatedTo2Sharing}([\![L]\!], \{P_3, P_1\})$$

2. Shuffle & Reshare to next shuffling pair #1:

   (a) P_1, P_2 let $[X']^{(1,2)} = [\pi_{\{1,2\}}(X)]^{(1,2)}$. Note that since P_i, P_j hold $[X]^{(i,j)}$ they can obtain $[\pi(X)]^{(i,j)}$ for a known π by locally shuffling their list of secret shares. P_1, P_2 reshare X' to the next shufflers, calling

$$[X'']^{(2,3)} = \mathcal{F}_{\text{ABB}}.\text{Reshare2To2Sharing}([X']^{(1,2)}, \{P_2, P_3\}).$$

   (b) P_3, P_1 let $[L']^{(3,1)} = [\pi^{-1}_{\{3,1\}}(L)]^{(3,1)}$. P_1, P_2 reshare L' to the next shufflers, calling

$$[L'']^{(2,3)} = \mathcal{F}_{\text{ABB}}.\text{Reshare2To2Sharing}([L']^{(3,1)}, \{P_2, P_3\})$$

3. Shuffle & Reshare next shuffling pair #2:

   (a) P_2, P_3 let $[X'']^{(2,3)} = [\pi_{\{2,3\}}(X')]^{(2,3)}$. P_2, P_3 reshare X'' to the next shufflers, calling

$$[X'']^{(3,1)} = \mathcal{F}_{\text{ABB}}.\text{Reshare2To2Sharing}([X']^{(2,3)}, \{P_3, P_1\}).$$

   (b) P_2, P_3 let $[L'']^{(2,3)} = [\pi^{-1}_{\{2,3\}}(L')]^{(2,3)}$. P_2, P_3 reshare L'' to the next shufflers, calling

$$[L'']^{(1,2)} = \mathcal{F}_{\text{ABB}}.\text{Reshare2To2Sharing}([L'']^{(1,2)}, \{P_1, P_2\}).$$

4. Shuffle & Reshare back to all players #3:

   (a) P_3, P_1 let $[X''']^{(3,1)} = [\pi_{\{3,1\}}(X'')]^{(3,1)}$. P_3, P_1 reshare X''' back to the entire group, calling

$$[\![X''']\!] = \text{Reshare2SharingToReplicated}([X''']^{(3,1)}).$$

   (b) P_1, P_2 let $[L''']^{(1,2)} = [\pi^{-1}_{\{1,2\}}(L'')]^{(1,2)}$. P_1, P_2 reshare L''' back to the entire group, calling

$$[\![L''']\!] = \text{Reshare2SharingToReplicated}([L''']^{(1,2)}).$$

Output: The output of the protocol is $[\![\pi(X)]\!], [\![\pi^{-1}(L)]\!]$.

Figure 5: The Persistent shuffle protocol, $\mathcal{F}_{\text{ABB}}.\text{ObliviousShuffle}([\![X]\!], \mathsf{DistributeShuffle} = \mathit{True})$

$\Pi_{\textbf{ShufTable}}$.**Build:** The players (refers to P_1, P_2, P_3) hold $([\![X_1]\!], [\![Y_1]\!]), \ldots, ([\![X_n]\!], [\![Y_n]\!])$.

1. The players create dummies to satisfy queries that were not found, letting $[\![Y_i]\!] = [\![\perp]\!]$, $[\![X_i]\!] = [\![\perp]\!]$ for $i \in \{n+1, \ldots, n+\mathsf{numDummies}\}$.
2. The players generate a κ-bit secret-shared PRF key, evaluating $[\![k]\!] = \mathcal{F}_{\text{ABB}}.\text{RandomElement}(\kappa)$.
3. The players create pseudorandom tags for all the addresses, evaluating $[\![Q_i]\!] = \mathcal{F}_{\text{ABB}}.\text{PRFEval}([\![k]\!], [\![X_i]\!])$ for $i \in \{1, \ldots, n\}$ and $[\![Q_i]\!] = [\![\perp]\!]$ for $i \in \{n+1, \ldots, n+\mathsf{numDummies}\}$.
4. Players obliviously shuffle the lists before revealing Q to P_1 to hide information about the elements in the table/stash, executing $[\![\hat{Q}]\!], [\![\hat{X}]\!], [\![\hat{Y}]\!], [\![j]\!] = \mathcal{F}_{\text{ABB}}.\text{ObliviousShuffle}([\![Q]\!], [\![X]\!], [\![Y]\!], \mathsf{DistributeShuffle} = \mathit{True})$ (c.f. Section 6).
   (a) The players locally create $[\![\mathsf{DI}_i]\!] = [\![j_{n+i}]\!]$ for all $i \in [\mathsf{numDummies}]$. If $\mathsf{DI}_i = k$, that implies $X_k = n+i$ and $Y_k = \perp$. That is, the k'th element of $(\hat{X}, \hat{Y})$ is the i'th dummy. We use DI in step 4 of query to return a dummy if needed.
5. Revealing Q_i to P_1 so that it can build a $\mathsf{CHT} \cup \mathsf{Stash}$ containing X_i using fast, local random accesses without learning X_i, the players call $\hat{Q}_i = \mathcal{F}_{\text{ABB}}.\text{RevealTo}(P_1, [\![\hat{Q}_i]\!])$ for $i \in \{1, \ldots, n+\mathsf{numDummies}\}$
6. P_1 locally constructs the Cuckoo hash table and list of stashed indices,s $\mathsf{CHT} \cup \mathsf{Stash} = \mathsf{BuildCHTwS}((\hat{Q}_i || i)_{i \in [n]}, \mathsf{stashSize})$. CHTstores $\hat{Q}_i || i$ for $n - \mathsf{stashSize}$ such different i's where $||$ represents bit-wise appending.
   (a) P_1 secret shares CHT between P_2 and P_3, running $[\mathsf{CHT}]^{(2,3)} = \mathcal{F}_{\text{ABB}}.\text{InputTo2Sharing}(\mathsf{CHT}, P_2, P_3)$. P_2, P_3 will use CHT to satisfy queries.
   (b) P_1 sends Stash, a cleartext list of stashSize-many indexes, s.t $i \in \mathsf{Stash}$ indicates that $([\![\hat{X}_i]\!], [\![\hat{Y}_i]\!])$ is stashed. If $i \in \mathsf{Stash}$ then $X_i \neq \perp$. **Output** $X^{\text{stash}} = \{[\![\hat{X}_i]\!]\}_{i \in \mathsf{Stash}}$ and $Y^{\text{stash}} = \{[\![\hat{Y}_i]\!]\}_{i \in \mathsf{Stash}}$. These will be reinserted into the cache when $\Pi_{\text{ShufTable}}$.Build is called as part Π_{DORAM} (Figure 8).
7. The players the data under under a permutation, $\hat{\pi}$, only known to P_2 and P_3, executing $[\![\hat{\hat{X}}]\!][\![\hat{\hat{Y}}]\!] = \mathcal{F}_{\text{ABB}}.\text{ObliviousShuffle}(([\![X]\!], [\![\hat{Y}]\!]), \text{RevealTo} = \{2,3\})$. This "rebalances the information asymmetry," allowing P_2, P_3 to guide P_1 to respond to queries (see step 5 of query) s.t. he cannot use his privileged information from $\Pi_{\text{ShufTable}}$.Build, not learning anything about the query.

$\Pi_{\textbf{ShufTable}}$.**Query:** The players hold $[\![Q_{query}]\!] = \mathcal{F}_{\text{ABB}}.\text{PRFEval}([\![X_t]\!], [\![k]\!])$ (Section 6.3). ***Each step corresponds to a single round of communication.*** We pack parallelizable/silent instructions into the same step.

1. First, the players compute $[\![r]\!] = \mathcal{F}_{\text{ABB}}.\text{RandomElement}(\kappa)$ (silent generation of random κ-bit secret share) and then compute $[\![q]\!] = [\![Q_{query}]\!] + [\![\mathsf{useDummy}]\!] \cdot [\![r]\!]$ (one MPC multiplication). useDummy is an artifact of the hierarchical solution that indicates if X_t was already found in previous level.
2. The players reveal enable P_2, P_3 to query $[\mathsf{CHT}]^{(2,3)}$ by revealing q to them, evaluating, $q = \mathcal{F}_{\text{ABB}}.\text{RevealTo}([\![q]\!], P_2, P_3)$.
3. P_2, P_3 input $[\mathsf{CHT}[h_1(q)]]^{(2,3)}, [\mathsf{CHT}[h_2(q)]]^{(2,3)}$ their secret-shares of locations in CHT where q might be stored, calling $[\![q'_b \mid\mid i'_b]\!] = \mathcal{F}_{\text{2Share}}.\text{Reshare2to3WithoutCheck}([\![T[h_b(Q)]]\!])$ for $b = 1, 2$.
4. The players evaluate QueryCircuit using (3,1) garbled circuits (see [33]), evaluating $l = \mathcal{F}_{\text{ABB}}.\text{EvalCircuit(3,1)GC}(\text{QueryCircuit}, \text{Inputs} = [\![q'_1]\!], [\![i'_1]\!], [\![q'_2]\!], [\![i'_2]\!], [\![q]\!], [\![\mathsf{DI}_t]\!])$, *revealing l only to P_2, P_3.* QueryCircuit returns i'_1 if $q'_1 = q$, returns i_2 if $Q'_2 = q$, and returns DI_t otherwise.
5. P_2, P_3 send $j = \pi_Y(l)$ to P_1. The parties set $[\![Y_{output}]\!] = [\![\hat{\hat{Y}}_j]\!]$ and append j to list queriedDblhatIdxs

$\Pi_{\textbf{ShufTable}}$.**Extract:** Output $([\![\hat{\hat{X}}_i]\!], [\![\hat{\hat{Y}}_i]\!])$ for $i \in ([n + \mathsf{numDummies}] - \text{queriedDblhatIdxs})$

Figure 6: $\Pi_{\text{ShufTable}}$.Build and $\Pi_{\text{ShufTable}}$.Query.

depth circuit, we can implement it using the 3-party MPC of [2] extremely efficiently. We give the SpeedCache protocol, $\Pi_{\text{SpeedCache}}$, in Figure 7.

E LowMC

In this section, we discuss LowMC, an MPC-friendly block-cipher we use to instantiate $\mathcal{F}_{\text{ABB}}$.PRFEval.

LowMC. LowMC (Low Multiplicative Complexity) [1] is a family of block cipher that is built with MPC, ZK, and FHE in mind. LowMC has a variety of instantiations which trades low number of AND gates and a low circuit AND-depth. [17] The instanstiation of LowMC we use has 46837 total gates, out of which 1134 are ANDs, stacked into 9-AND-depth circuit. By contrast, AES has a total of 36663 gates, out of which 6400 are ANDs, stacked into a 60-AND-depth-circuit. [18] Since the AND-depth of the circuit is equal to the number of rounds to evaluate the circuit under MPC, using LowMC instead of AES saves 51 rounds of communication for each query.

We provide the first circuit files for LowMC and encode them in the popular bristol fashion [19] format. The traditional bristol fashion format requires that each virtual "wire" to be only assigned once, requiring significantly more memory for the computation than necessery. We show that by threading wires through the circuit (i.e. reusing memory) we are able speed up the computation of LowMC under the [2] MPC by a factor of 2.

Although LowMC has not received as much analysis as AES, it's security has been been adapted into the Picnic signature scheme [10], a 3rd round candidate in the NIST post-quantum digital signature contest [35]. Additionally, there have been several thorough cryptanalysis attempts [5, 15, 31] motivated by an ongoing Microsoft-funded challenge, [20] none of which discovered an attack which made the community doubt the security LowMC.

F Alibi reinsertion

In this section, we outline our instantiation of Alibi reinsertion, a "cache the stash" technique originally presented in [18] As discussed in Section 6.2, since we use Cuckoo hash tables to store ShufTable's data, we must reduce the failure probability of $\Pi_{\text{ShufTable}}$.Build from noticeable to negligable (in N) by outputting a small stash, Stash, which we reinsert into the cache. Naively, for all $(\llbracket X_i \rrbracket, \llbracket Y_i \rrbracket) \in \text{Stash}$ we could call L_0.Append$(\llbracket X_i \rrbracket, \llbracket Y_i \rrbracket)$, "reinserting" $(\llbracket X_i \rrbracket, \llbracket Y_i \rrbracket)$ which could not fit in L_i's CHT into the cache. Yet, as proved in [18], the naive reinsertion technique above comparmises the obliviousness of (D)ORAM. Intuitively, if we reinsert $(\llbracket X_i \rrbracket, \llbracket Y_i \rrbracket)$ to L_0 from L_j and then query $\llbracket X_i \rrbracket$, in the aggragate, an eavsedropper could tell that we should have queried $\llbracket X_i \rrbracket$ to L_j, but instead queried a dummy becuase we found $(\llbracket X_i \rrbracket, \llbracket Y_i \rrbracket)$ at some previous level (it was reinserted). Roughly speaking, our goal $(*)$ is to append some data to $(\llbracket X_i \rrbracket, \llbracket Y_i \rrbracket)$ such that when $(\llbracket X_i \rrbracket, \llbracket Y_i \rrbracket)$ is reinserted from L_j and found at L_k for $k < j$, we will know to continue querying $L_{k+1}, \ldots, L_j$ *as if* we did not find $(\llbracket X_i \rrbracket, \llbracket Y_i \rrbracket)$, and query $L_{j+1}, \ldots, L_{\text{numLevels}}$ *as if* we found $(\llbracket X_i \rrbracket, \llbracket Y_i \rrbracket)$ at L_j. To do this, we store $\llbracket \mathfrak{e}_{X_i} \rrbracket \in \{0,1\}^{\text{numLevels}}$ as the last numLevels bits of Y_i (it is assumed that $D >$ numLevels) where $\mathfrak{e}_{X_i}[j] = 1$ iff $(\llbracket X_i \rrbracket, \llbracket Y_i \rrbracket)$ was reinserted from L_j. We set $\mathfrak{e}_{X_i} = 0^{\text{numLevels}}$ every time we query $(\llbracket X_i \rrbracket, \llbracket Y_i \rrbracket)$. When querying for $\llbracket x \rrbracket$ and finding it at L_k where $\mathfrak{e}_x[j] = 1$ from L_j, under MPC we "decide" to query according to $(*)$. We present the full details in Figure 8.

G Full DORAM protocol Π_{DORAM}

We give the GigaDORAM protocol, Π_{DORAM}, in Figure G.

H Parameter instantiation

Depth of the hierarchy. We fix all of our variables as functions of the total number of records, N, the size of each record, D, and the security parameter κ. With these inputs, we experiment with different values of baseAmpFactor, stashSize and numLevels to find the number of levels that minimizes running time. Throughout our testing, we set $|L_0| \approx 2^{10}$, numLevels ≈ 5 and adjust baseAmpFactor across N to accomodate.

Cuckoo hash table parameters. We set the number of slots per table $c = 1.2n$ and number of tables $t = 2$. To ensure that the CHTs have negligible probability of build failure, we need to include a stash [27]. Theoretical analyses [36] give an bound on the size of a stash necessary to ensure a negligible probability of build failure. In practice, however, much smaller stash sizes appear to give vanishingly small probability of build failure [41]. For our implementation, we have a simple configuration variable that allows us to switch between these two methods of determining the stash size. In our benchmarking, we use the (smaller) empirical bounds, setting stashSize $= 8$, and in all of our testing, we never recorded a stash with more than 4 elements.

[17] The authors of LowMC provide a script to exactly calculate the tradeoff between the number of AND gates, and the AND-depth of the circuit https://github.com/LowMC/lowmc/blob/master/determine_rounds.py.

[18] We refer to Bristol Fashion AES circuit file from https://homes.esat.kuleuven.be/~nsmart/MPC/MAND/aes_128.txt

[19] https://homes.esat.kuleuven.be/~nsmart/MPC/

[20] https://lowmcchallenge.github.io/

($\star$) symbolizes the invarient that when $\Pi_{\text{SpeedCache}}$.Query is called $i \neq j \implies x_i \neq x_j$.
$\Pi_{\textbf{SpeedCache}}$.**Init**()**:** The players set the query counter $t = 0$. No communication.
$\Pi_{\textbf{SpeedCache}}$.**Store**($[\![x]\!], [\![y]\!]$)**:** Let $[\![x_t]\!] = [\![x]\!]$ and $[\![y_t]\!] = [\![y]\!]$. No communication.
$\Pi_{\textbf{SpeedCache}}$.**Query**($[\![x]\!]$)**:** $([\![x_1]\!], [\![y_1]\!]), \ldots, ([\![x_t]\!], [\![y_t]\!])$ are stored where x_i is unique. Note that this invariant, ($\star$), inductively holds: On Init ($\star$) trivially holds. On Store(x) at the end of Π_{DORAM}.ReadAndWrite, if x was stored at L_0, it was zeroed-out in step 3 below and thus ($\star$) holds. On alibi-reinsertion from L_i since L_i.Build($[\![X]\!], [\![Y]\!]$), ($\star$) holds becuase X is gurenteed to be a unique list and thus its reinserted subset also unique.

1. Using MPC, the players compute equality-indicators $[\![b_i]\!]$ for all $i \in [t]$ s.t $b_i = 1$ iff $x = x_i$. This takes $\lceil \log \log N \rceil$ rounds and $t \log N$ communication, because each x_i is $\log N$ bits.

2. Using MPC, the players zero-out all the non-queried elements, computing the bit-vector product $[\![x'_i]\!] = [\![x'_i]\!] \cdot [\![b_i]\!]$ and $[\![y'_i]\!] = [\![y_i]\!] \cdot [\![b_i]\!]$ for all $i \in [t]$. This takes a single round and $n(\log N + D)$ communication.

3. The players zero-out (put a dummy) at the location where x_i was found by setting $[\![x_i]\!] := [\![x_i]\!] \oplus [\![x'_i]\!]$ for all $i \in [n]$. Then, they output $\bigoplus_{i=1}^{n} [\![y'_i]\!]$ which is $[\![y_i]\!]$ if $x = x_i$, else $[\![\perp]\!]$ This costs no communication or rounds.

$\Pi_{\textbf{SpeedCache}}$.**Extract:** Output $([\![x_1]\!], [\![y_1]\!]), \ldots, ([\![x_t]\!], [\![y_t]\!])$, set $t = 0$. No communication.

Figure 7: The SpeedCache protocol

$\Pi_{\mathbf{DORAM}}$.Init($[\![Y]\!]$):

1. The players input the address of each payload $[\![X_i]\!] = \mathcal{F}_{\text{ReplicatedMPC}}.\text{InputConstant}(i)$ for $i = 1$ to N.
2. The players build the toplevel $L_{\mathsf{numLevels}} = \mathcal{F}_{\text{OHTable}}.\text{Build}([\![X]\!], [\![Y]\!], N)$ and initialize the cache $L_0.\text{Init}()$. The players set the query counter $t = 0$.

$\Pi_{\mathbf{DORAM}}$.ReadAndWrite($[\![X_{\mathbf{query}}]\!]$,$[\![Y_{\mathbf{new}}]\!]$, $[\![\text{isWrite}]\!]$): $|L_0|$ is the size of the cache and is tracked externally.

1. The players increment the query counter, t, initialize numLevels-bit Alibi data accumalator $[\![\mathfrak{e}_{\text{accum}}]\!] = [\![0^{\mathsf{numLevels}}]\!]$, D-bit payload accumalator $[\![Y_{\text{accum}}]\!] = [\![\perp]\!]$, and one-bit flag $[\![\mathsf{found}]\!] = 0$. Then, the players query the Cache, $[\![Y_{\text{accum}}]\!], [\![\mathsf{found}]\!] = L_0.\text{Query}([\![X_{\text{query}}]\!])$. The players extract $[\![\mathfrak{e}_{\text{accum}}]\!]$ from $[\![Y_{\text{accum}}]\!]$ (by silently copying the last numLevels-bits). This step takes numRoundsCache rounds.
2. For each ℓ from 1 to numLevels, if there is an OHTable at level L_ℓ:
   (a) Silently compute the bit $[\![\mathsf{useDummy}]\!] = [\![\mathfrak{e}]\!][\ell] \oplus [\![\mathsf{found}]\!]$ and then query L_i, calling $[\![Y_\ell]\!], [\![\mathsf{found}_\ell]\!] = L_\ell.\text{Query}([\![X]\!], [\![\mathsf{useDummy}]\!])$. This step takes numRoundsOHTable rounds.
   (b) Silently extract the numLevels-bit value $[\![\mathfrak{e}_\ell]\!]$ from $[\![Y_\ell]\!]$ and update $[\![\mathfrak{e}_{\text{accum}}]\!] \leftarrow [\![\mathfrak{e}_{\text{accum}} \oplus \mathfrak{e}_\ell]\!]$. Additionally, In parallel with the first round of the next iteration of the loop, update $[\![Y_{\text{accum}}]\!] \leftarrow \mathcal{F}_{\text{ReplicatedMPC}}.\text{IfThenElse}([\![\mathsf{found}]\!], [\![Y_\ell]\!], [\![Y_{\text{accum}}]\!])$. This step takes no rounds.
3. Set $[\![Y_{\text{new}}]\!] \leftarrow \mathcal{F}_{\text{ReplicatedMPC}}.\text{IfThenElse}([\![\text{isWrite}]\!], [\![Y_{\text{new}}]\!], [\![Y_{\text{accum}}]\!])$. Set the Alibi mask of $[\![Y_{\text{new}}]\!]$, $[\![Y_{\text{new}}.\mathfrak{e}]\!]$ to 0. Call $\mathcal{F}_{\text{Linear}}.\text{Store}([\![X]\!], [\![Y_{\text{new}}]\!])$. This takes 1 round.
4. If $t = |L_0|$, run the subroutine $\Pi_{\text{DORAM}}.\text{Rebuild}()$.

$\Pi_{\mathbf{DORAM}}$.Rebuild(): We define a level L_i for $1 \le i \le \mathsf{numLevels}$ in the DORAM to be *full* if it contains a ShufTable with $(\mathsf{baseAmpFactor} - 1) \cdot \mathsf{baseAmpFactor}^i \cdot |L_0|$ elements. L_0 is full if it has been written to $|L_0|$ times since it was last initialized. Suppose that ℓ is the largest number such that levels $L_0, \ldots L_\ell$ are full while $L_{\ell+1}$ is not, instead containing $A\mathsf{baseAmpFactor}^\ell |L_0|$ elements for some $A \in \{0, \ldots, \mathsf{baseAmpFactor} - 2\}$ If $\ell = \mathsf{numLevels}$, then necesserily $A = 1$.

1. By concatenating the output of $L_0.\text{Extract}()$ and $L_i.\text{Extract}()$ for $i \in [\ell + 1]$, the players prepere lists $[\![X]\!]$, $[\![Y]\!]$ of length $(A+1) \cdot \mathsf{baseAmpFactor}^\ell \cdot |L_0|$ containing all the elements to be (potentially) placed in $L_{\ell+1}$.
2. If $\ell < \mathsf{numLevels}$:
   (a) In parallel, relabel the dummies, calling $[\![X_j^*]\!] \leftarrow \mathcal{F}_{\text{ReplicatedMPC}}.\text{ReplaceIfNull}([\![X_j]\!], [\![N + j]\!])$ and $[\![Y_j^*]\!] = [\![Y_j]\!]$ (syntactically convinient) for $j \in [(A+1) \cdot \mathsf{baseAmpFactor}^\ell \cdot |L_0|]$ (this is so P_1 does not learn how many dummies were queried at the previous level).
3. If $\ell = \mathsf{numLevels}$:
   (a) The players "cleanse out the dummies" by shuffling, $[\![\hat{X}]\!], [\![\hat{Y}]\!] = \mathcal{F}_{\text{ABB}}.\text{ObliviousShuffle}([\![X]\!], [\![Y]\!])$, and then revealing which element is dummy by calling $\mathcal{F}_{\text{ABB}}.\text{Reveal}(\mathcal{F}_{\text{ABB}}.\text{Equals}([\![\hat{X}_j]\!], \perp))$ for all j.
   (b) The above will reveal exactly N 0's and N 1's (because we started with N "real" elements which we have maintained and we made N queries since last build $L_{\mathsf{numLevels}}$. Since stale elements are relabeled (step 2.A) the invarient holds). Compact the $[\![\hat{X}]\!]$ and $[\![\hat{Y}]\!]$ shares corresponding to 0's into arrays $[\![X^*]\!]$ and $[\![Y^*]\!]$.
4. The players reintialize the cache by calling $L_0 = \mathcal{F}_{\text{Linear}}.\text{Init}(|L_0|)$ and build $L_{\ell+1}$ by calling $[\![X^{\text{stash}}]\!], [\![Y^{\text{stash}}]\!] = L_{\ell+1}.\text{Build}([\![X^*]\!], [\![Y^*]\!])$
5. For i from 1 to stashSize, update the Alibi bit $[\![\mathfrak{e}]\!][\ell + 1] =$ of $[\![Y_i^{\text{stash}}]\!]$ to 1, then call $\mathcal{F}_{\text{Linear}}.\text{Store}([\![X_i^{\text{stash}}]\!], [\![Y_i^{\text{stash}}]\!])$.
6. Reset $t = 0$.

Figure 8: $\Pi_{\text{ShufTable}}.\text{Build}$ and $\Pi_{\text{ShufTable}}.\text{Query}$.